FreeBSD™ JOURNAL

Jan/Feb 2015

BRINGING UP MIPS

The New 64-bit Core Design Architecture

arm64

NEW!

INTERACTING

- Getting Involved with the FreeBSD Project
- A Journey from Adoption to Contribution

pfSense

2.2

FreeBSD 10.1-based Network Security
from the pfSense® team.

Cloud Solutions:

vmware® READY

powered by amazon web services

Hardware Appliances:

AES-NI Acceleration for IPSec

Professional Support

amazon web services VPC wizard

Exclusively available from Netgate, the home of pfSense.

Netgate®

7212 McNeil Drive Suite 204 Austin, TX 78729 +1.512.646.4100

Table of Contents

Jan/Feb 2015

FreeBSD JOURNAL Vol. 2, Issue No. 1

FreeNAS

in an Enterprise Environment

By the time you're reading this, FreeNAS has been downloaded more than 5.5 million times. For home users, it's become an indispensable part of their daily lives, akin to the DVR. Meanwhile, all over the world, thousands of businesses universities, and government departments use FreeNAS to build effective storage solutions in myriad applications.

What you will learn...

- How TrueNAS builds off the strong points of the FreeBSD and FreeNAS operating systems
- How TrueNAS meets modern storage challenges for enterpri

The FreeNAS operating systems is freel the public and offers thorough docu active community, and a feature-rich the storage environment. Based on FreeB can share over a host of protocols (SMB, FTP, iSCSI, etc) and features an intuitive the ZFS file system, a plug-in system fo much more.

Despite the massive popularity of aren't aware of its big brother dutif data in some of the most demandin environments: the proven, enterpri professionally-supported line of a

But what makes TrueNAS differ Well, I'm glad you asked...

Commercial Grade Suppo

When a mission critical stora organization's whole operati halt. Whole community-base free), it can't always get an and running in a timely ma responsiveness and expert dedicated support team provide that safety.

Created by the same developed FreeNAS.

NEW RELEASE

iX systems®

WE INTERRUPT THIS MAGAZINE TO BRING YOU THIS IMPORTANT ANNOUNCEMENT:

THE PEOPLE WHO DEVELOP FREENAS, THE WORLD'S MOST POPULAR STORAGE OS, HAVE JUST REVAMPED TRUENAS.

POWER WITHOUT CONTROL MEANS NOTHING. TRUENAS STORAGE GIVES YOU BOTH.

- ☑ Simple Management
- ☑ Hybrid Flash Acceleration
- ☑ Intelligent Compresssion
- ☑ All Features Provided Up Front (no hidden licensing fees)
- ☑ Self-Healing Filesystem
- ☑ High Availability
- ☑ Qualified for VMware and HyperV
- ☑ Works Great With Citrix XenServer®

To learn more, visit: www.iXsystems.com/truenas

intel® Xeon® inside™

POWERED BY INTEL® XEON® PROCESSORS

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher •	Walter Andrzejewski walter@freebsdjournal.com
Editor-at-Large •	James Maurer jmaurer@freebsdjournal.com
Art Director •	Dianne M. Kischitz dianne@freebsdjournal.com
Office Administrator •	Michael Davis davism@freebsdjournal.com
Advertising Sales •	Walter Andrzejewski walter@freebsdjournal.com Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).

Published by the FreeBSD Foundation, PO Box 20247, Boulder, CO 80308
ph: 720/207-5142 • fax: 720/222-2350
email: board@freebsdfoundation.org
Copyright © 2015 by FreeBSD Foundation. All rights reserved.

This magazine may not be reproduced in whole or in part without written permission from the publisher.

Welcome to *FreeBSD Journal* 2015!

With this new year, we bring our *FreeBSD Journal* subscribers a fresh slate of articles, as well as a couple of new developments, the first of which is the inclusion of articles about interacting with the FreeBSD Project. In this issue Dru Lavigne writes about different ways to get involved, and Rick Miller and Julien Charbon discuss how they went from adopting FreeBSD within Verisign to being active contributors to the FreeBSD source base.

One of the *Journal*'s goals is to serve as a storehouse of knowledge about FreeBSD and associated software systems. It has always been the Board's intention to release some of the *Journal*'s articles online—after a reasonable period of time—so that they can be indexed, referenced, and used by the general public. Beginning immediately, we will start that process by releasing a few of the articles from the Jan/Feb 2014 issue on the FreeBSD Foundation's website. If you are a subscriber, know that your subscription gets you the articles well ahead of any free releases plus the timely svn update and Ports Report columns, book reviews, conference reports, calendar, and This Month in FreeBSD, all of which will not be part of the free release.

FreeBSD Journal's circulation continues to grow each month and we expect that to continue in 2015. You can help us by showing other folks our work, posting pointers to articles, and making sure everyone you know who should know about FreeBSD also knows that the *FreeBSD Journal* is the best way to keep up with the latest releases and new developments in FreeBSD.

Finally, for European readers on iOS, please check your subscriptions. An increase in VAT within Europe caused Apple to cancel all automatic magazine subscription renewals, meaning that you will need to re-subscribe to the *FreeBSD Journal* if you bought it on iTunes from one of the European stores—even if you checked automatic renewal.

Trust us, no one wants to miss even one of our issues in 2015.

Sincerely,
FreeBSD Journal Editorial Board

bringing up MIPS

Porting FreeBSD to a new CPU, even within a previously supported family, is a significant undertaking.

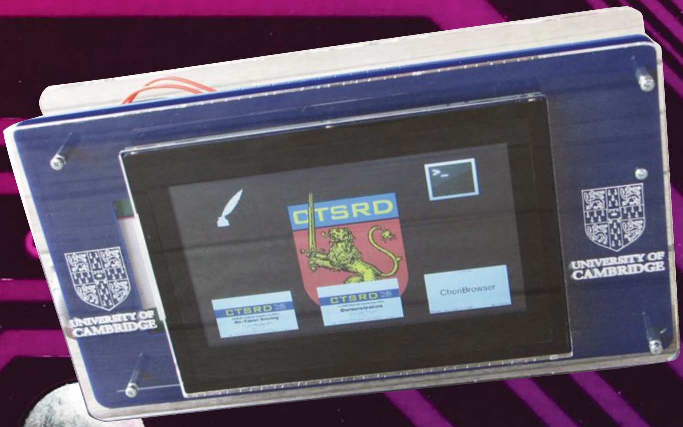


Fig. 1 BERIpad with Application Launcher.

By Brooks Davis, Robert Norton, Jonathan Woodruff & Robert N. M. Watson

We hope this article will help prospective porters orient themselves before they begin the process. While we focus on MIPS ports, the code structure in other platforms—especially ARM—is quite similar. From its modest origins as a fork of 386BSD targeting Intel i386 class CPUs, FreeBSD has been ported to a range of architectures including AArch64, AMD x86_64 (aka amd64), ARM, MIPS, and PowerPC. While the x86 systems are fairly homogeneous targets with mechanics for detecting and adapting to specific board and peripheral configurations, embedded systems platforms such as ARM, MIPS, and PowerPC are much more diverse. Porting to a new MIPS board often requires adding support for a new system on chip (SoC) or CPU type with different interrupt controllers, buses, and peripherals. Even if the CPU is supported, boot loaders and associated kernel calling conventions generally differ significantly between boards.

We have ported FreeBSD/MIPS to BERI, an open-source MIPS R4000-style [1] FPGA-based soft-core processor that we have developed. This required a range of work including boot-loader support, platform startup code, a suite of device drivers (including the PIC), and also adapting FreeBSD's existing FDT support to FreeBSD/MIPS. We currently run FreeBSD/BERI under simulation, on the Altera Stratix IV FPGA on a Terasic DE4 FPGA board, on the Altera Cyclone V on a Terasic SoCKit board, and on a Xilinx Virtex-5 FPGA on the NetFPGA-10G platform. The majority of our peripheral work has been in simulation and the DE4 platform. FreeBSD BERI CPU support is derived from the MALTA port, with some inspiration from the SiByte port.

Based on our experiences bringing up FreeBSD on BERI, we have documented how we boot FreeBSD from the firmware embedded in the CPU to userspace, to provide a new view on the boot process and help porters gain a high-level understanding of the boot process.

The rest of this article narrates the boot process with a special focus on the places where customization was required for BERI. In the interest of brevity, many aspects of boot are omitted; others that are not platform- or architecturally-specific are ignored. Some platform-specific components such as the MIPS pmap are not covered. The goal is to provide a guide to those pieces someone would need to know when porting to a new, but relatively conventional, MIPS CPU. Porters inter-

```
model = "SRI/Cambridge BeriPad (DE4)";
compatible = "sri-cambridge,beripad-de4";
cpus {
  cpu@0 {
    device-type = "cpu";
    compatible = "sri-cambridge,beri";
  };
};
soc {
  memory {
    device_type = "memory";
    reg = <0x0 0x40000000>;
  };
  beripic: beripic@7f804000 {
    compatible = "sri-cambridge,beri-pic";
    interrupt-controller;
    reg =  <0x7f804000 0x400 0x7f806000 0x10
      0x7f806080 0x10 0x7f806100 0x10>;
  }
  serial@7f002100 {
    compatible = "ns16550";
    reg = <0x7f002100 0x20>;
  };
  serial@7f000000 {
    compatible = "altera,jtag_uart-11_0";
    reg = <0x7f000000 0x40>;
  };
  sdcard@7f008000 {
    compatible = "altera,sdcard_11_2011";
    reg = <0x7f008000 0x400>;
  };
  flash@74000000 {
    partition@20000 {
      reg = <0x20000 0xc00000>;
      label = "fpga0";
    };
    partition@1820000 {
      reg = <0x1820000 0x027c0000>;
      label = "os";
    };
  };
  ethernet@7f007000 {
    compatible = "altera,atse";
    reg = <0x7f007000 0x400 0x7f007500 0x8
      0x7f007520 0x20 0x7f007400 0x8
      0x7f007420 0x20>;
  };
  touchscreen@70400000 {
    compatible = "sri-cambridge,mtl";
    reg = <0x70400000 0x1000
      0x70000000 0x177000 0x70177000 0x2000>;
  };
};
```

Fig. 2 Excerpt from Flat Device Tree (FDT) Description of the DE4-based BERI Tablet.

ested in less conventional CPUs will want to examine the NLM and RMI ports in `mips/nlm` and `mips/rmi` for examples of the more extensive modifications required for a complete multi-core platform.

For more information on the high-level boot process see Chapter 15 of *The Design and Implementation of the FreeBSD Operating System, Second Edition* [7].

The BERIpad Platform

We have developed BERI as a platform that can enable experiments on the hardware-software interface such as our ongoing work on hardware-supported capabilities in the CHERI CPU [6]. Our primary hardware target has been a tablet based on the Terasic DE4 FPGA board with a Terasic MTL touch screen and integrated battery pack. The source for the CPU and the design for the tablet have been released as open source at http://beri-cpu.org. The modifications to FreeBSD support have been merged to FreeBSD. Single-processor support appeared in 10.0 and multiprocessor support will appear in future releases. The tablet (Figure 1) and the internal architecture of BERI are described in detail in The BERIpad Tablet [2].

We developed device drivers for three Altera IP cores: the JTAG UART (`altera_jtag_uart`), triple-speed MAC (`atse`), and SD Card (`altera_sdcard`), which implement low-level console/tty, Ethernet interface, and block storage classes. In addition, we have implemented a generic driver for Avalon-attached devices (`avgen`) that allows memory mapping of arbitrary bus-attached devices without interrupt sources—such as the DE4 LED block, BERI configuration ROM, and DE4 fan-and-temperature control block.

Flat Device Tree

Most aspects of BERI board configuration are described in Flat Device Trees (FDTs), which are commonly used on PowerPC and ARM-based systems [3]. Currently a Device Tree Blob (DTB) is built into each FreeBSD kernel, describing a specific hardware configuration. Each DTB is built from a device-tree syntax (DTS) file by the device-tree compiler (`dtc(1)`) before being embedded in the kernel. Figure 2 excerpts `boot/fdt/dts/mips/beripad-de4.dts`—the DTS file—and includes the BERI CPU, 1GB DRAM, programmable interrupt controller (PIC), hardware serial port, JTAG UART, SD card reader, flash partition table, gigabit Ethernet, and touch screen.

The Early Boot Sequence

The common FreeBSD boot sequence begins with CPU firmware arranging to run the

FreeBSD boot2 second-stage boot loader, which in turn loads `/boot/loader`, which loads the kernel and kernel modules. This leads to the kernel boot, described later in this article in a section called "The Path to Usermode."

Miniboot

At power on or after reset, the CPU sets the program counter of at least one hardware thread to the address of a valid program. From the programmer perspective, the process by which this occurs is essentially magic and of no particular importance. Typically the start address is some form of read-only or flash-upgradable firmware that allows for early CPU setup and that may handle details such as resetting cache state or pausing threads other than the primary thread until the operating system is ready to handle them. In many systems, this firmware is responsible for working around CPU bugs.

On BERI this code is known as `miniboot` for physical hardware and `simboot` for simulation. `Miniboot` is compiled into the FPGA bitfile as a read-only BRAM. It is responsible for setting registers to initial values, setting up an initial stack, initializing the cache by invalidating the contents, setting up a spin table for multiprocessor (MP) boot, and loading the next-stage loader or kernel from flash, or waiting for it to be loaded via the debug unit and executing it. With BERI, we are fortunate that we need not work around CPU bugs in firmware, because we can simply fix the hardware.

`Miniboot` kernel loading and boot behavior is controlled by two DIP switches on the DE4. If DIP0 is off or if miniboot is compiled with `-DALWAYS_WAIT`, then we spin in a loop waiting for the MIPS-ISA general-purpose register `t1` to be set to `0` using JTAG. This allows a user to control when the processor starts executing, giving the user an opportunity to load a kernel directly to DRAM before boot proceeds. DIP1 controls the relocation of a kernel from flash. If the DIP switch is set, the kernel is loaded from flash at offset of `0x2000000` to `0x100000` in DRAM. Otherwise, the user must load a kernel via the debug unit as described in the BERI Software Reference [5].

The kernel will be loaded only on hardware thread 0. In other hardware threads, `miniboot` enters a loop waiting for the operating system to provide a kernel entry point via the spin-table. Booting with multithreading and multicore configurations is discussed later in this article in the section titled "Multiprocessor

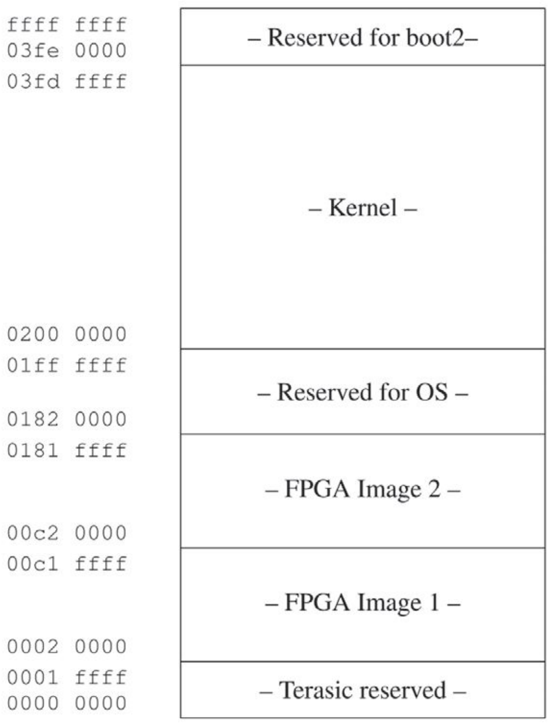


Fig. 3 Layout of the DE4 Flash.

Support."

Before `miniboot` enters the kernel, it clears most registers and sets `a0` to `argc`, `a1` to `argv`, `a2` to `env`, and `a3` to the size of system memory. In practice `argc` is `0` and `argv` and `env` are `NULL`. It then assumes that an ELF64 object is located at `0x100000`, loads the entry point from the ELF header, and jumps to it.

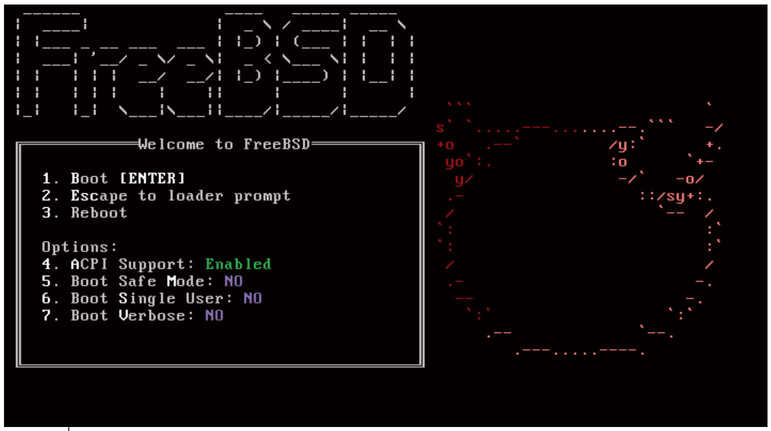
We intend for `miniboot` to be minimal, but also sufficiently flexible to support debugging of various boot layouts, as well as loading alternative code such as self-contained binaries. This allows maximum flexibility for software developers who may not be equipped to generate new hardware images.

boot2

On most FreeBSD systems two more boot stages are interposed between the architecture-dependent boot code and the kernel. The first of these is `boot2`, the second-stage bootstrap (`boot(8)`); `boot2` has a mechanism for accessing local storage and has code for read-only access to a limited set of file systems (currently UFS or ZFS). Its primary job is to load the loader and to pass arguments to it. By default it loads `/boot/loader`, but the user can specify an alternative disk, partition, and path. `boot2` can also boot the kernel directly.

We have ported `boot2` to BERI, creating three microdrivers—allowing JTAG UART console access, and use of CFI or the SD card to load `/boot/loader` or the kernel. These microdrivers substitute for boot device drivers provided by the BIOS on x86 or OpenFirmware on SPARC. It also supports jumping to an instance of `/boot/loader` loaded via JTAG. In our current implementation, `boot2` is linked at the same address as the kernel and loaded from CFI flash allowing it to be used with an unmodified `miniboot`. In the future, we plan to place a similar version of `boot2` at `0x03fe0000`, a 128K area reserved for its use. This will allow a normal file system to be placed in CFI flash from `0x1820000`, which might contain the full boot loader, a kernel, etc. Currently, we use `boot2` to relocate FDT embedded in the CPU image and to load `/boot/loader` from the SD card, which offers an experience more akin to conventional desktop/server platforms than conventional embedded targets.

Many versions of `boot2` exist that are targeted at different architectures. The version of `boot2` in BERI is derived from the x86 boot2,



```
Welcome to FreeBSD

 1. Boot [ENTER]
 2. Escape to loader prompt
 3. Reboot

 Options:
 4. ACPI Support: Enabled
 5. Boot Safe Mode: NO
 6. Boot Single User: NO
 7. Boot Verbose: NO
```

Fig. 4 FreeBSD Loader Boot Menu

and is (marginally) more feature-rich than those targeted at more space-constrained embedded architectures. There is currently more diversity of `boot2` implementations than necessary, largely due to excessive copying and modifying of code during the early parts of the porting process.

loader

The third common boot stage is the `loader(8)`. The loader is in practice a small kernel whose main job is to set up the environment for the kernel and then load the kernel and any configured modules from the disk or network. The loader contains a Forth interpreter based on FICL (http://ficl.sourceforge.net). This interpreter is used to provide the boot menu shown in Figure 4; it parses configuration files such as /boot/loader.conf, and implements functionality such as `nextboot(8)`. In order to do this, the loader also contains microdrivers to access platform-specific devices and contains implementations of UFS and ZFS with read and, limited, write support. On x86 systems that means BIOS or UEFI disk access, and with the pxeloader network access via PXE. On BERI this currently includes a basic microdriver for access to the CFI flash found on the DE4.

We have ported the loader to FreeBSD/MIPS and shared the SD card and CFI microdrivers with `boot2` to allow kernels to be loaded from CFI flash or SD card. We currently load the kernel from the SD card. We hope to eventually add a driver for the on-board Ethernet device to allow us to load kernels from the network.

The loader's transition to the kernel is much the same as miniboot. The kernel is loaded to the expected location in the memory, the ELF header is parsed, arguments are loaded into registers, and the loader jumps into the kernel.

The bootinfo Structure

In order to facilitate passing information among `boot2`, `/boot/loader`, and the kernel, a pointer to a `bootinfo` structure allows information such as memory size, boot media type, and the locations of preloaded modules to be shared. When present and detected by `boot2`, the location in memory of a relocated copy of embedded FDT is also present in `bootinfo`.

The Path to Usermode

This section narrates the interesting parts of the FreeBSD boot process from a MIPS porter's perspective.

Early Kernel Boot

The FreeBSD MIPS kernel enters at `_start` in the `_locore` function defined in `mips/mips/locore.S`; `_locore` performs some early initialization of the MIPS CP0 registers, sets up an initial stack, and calls the platform-specific startup code in `platform_start`.

On BERI, `platform_start` saves the argument list, environment, and pointer to `struct bootinfo` passed by the loader. BERI kernels also support an older boot interface, in which memory size is passed as the fourth argument (direct from `miniboot`). It then calls the common MIPS function `mips_postboot_fixup`, which provides kernel-module information for manually loaded kernels and corrects `kernel_kseg0_end` (the first usable address in kernel space) if required. Per-CPU storage is then initialized for the boot CPU by `mips_pcpu0_init`. Since BERI uses Flat Device Tree to allow us to configure otherwise non-discoverable devices, `platform_start` then locates the DTB and initializes FDT. This is the norm for ARM and PowerPC ports, but is currently uncommon on MIPS ports. We expect it to become more popular over time. The `platform_start` function then calls `mips_timer_early_init` to set system timer constants, currently to a hardcoded 100MHz—although eventually this will come from FDT. The console is set up by `cninit` and some debugging information is printed. The number of pages of real memory is stored in the global variable `realmem`[1]. The BERI-specific `mips_init`[2] function is then called to perform the bulk of the remaining early setup.

BERI's `mips_init` is fairly typical. First, memory-related parameters are configured including laying out the physical memory range and setting a number of automatically tuned parameters in the general functions `init_param1` and `init_param2`. The MIPS function `mips_cpu_init` performs some optional per-platform setup (nothing on BERI), identifies the CPU, configures the cache, and clears the TLB. The MIPS version of `pmap_bootstrap` is called to initialize the pmap. Thread 0 is instantiated by `mips_proc0_init`, which also allocates space for dynamic per CPU variables. Early mutexes (including the legacy Giant lock) are initialized in `mutex_init`, and the debugger is initialized in `kdb_init`. If so configured, the kernel may now drop into the debugger or, much more commonly, return and continue booting.

Finally `mips_timer_init_params` is called to finish setting up the timer infrastructure before `platform_start` returns to `_locore`, which switches to the now configured `thread0` stack and calls `mi_startup`, never to return.

Calling All SYSINITS

The job of `mi_startup` is to initialize all the kernel's subsystems in the right order. Historically, `mi_startup` was called `main` and the order of initialization was hard-coded. This was obviously not scalable, so a more dynamic registration mechanism called `SYSINIT(9)` was created. Any code that needs to be run at start-up can use the `SYSINIT` macro to cause a function to be called in a sorted order to boot or on module load. The sysinit implementation relies on the *linker set* feature, by which constructors/destructors for kernel subsystems and modules are tagged in the ELF binary so that the kernel linker can find them during boot, module load, module unload, and kernel shutdown.

The implementation of `mi_startup` is simple. It sorts the set of sysinits and then runs each in turn marking each as done when it is complete. If any modules are loaded by a sysinit, it re-sorts the entire set and starts from the beginning, skipping previous run entries. The end of `mi_startup` contains code to call `swapper`, which is never reached—as the last sysinit never returns. One implementation detail of note in `mi_startup` is the use of bubble sort to sort the sysinits, due to the fact that allocators are initialized via sysinits and thus not yet available.

Figure 5 shows a simple example of a sysinit. In this example, `announce` is the name of the individual sysinit, `SI_SUB_COPYRIGHT` is the subsystem, `SI_ORDER_FIRST` is the order within the subsystem, `print_caddr_t` is the function to call, and `copyright` is an argument to be passed to the function. A complete list of subsystems and orders within subsystems can be found in `sys/kernel.h`. As of this writing there are more than 80 of them. Most have little or no architecturally-specific function, and thus are beyond the scope of this aticle. We highlight sysinits with significant port-specific content.

```
static void
print_caddr_t(void *data)
{
        printf("%s", (char *)data);
}
SYSINIT(announce, SI_SUB_COPYRIGHT,
   SI_ORDER_FIRST, print_caddr_t,
   copyright);
```

Fig. 5 Implementation of Copyright Message Printing on FreeBSD Boot.

The first sysinit of interest is `SI_SUB_COPYRIGHT`. It does not require porting specifically, but reaching it and seeing the output is a sign of an architectural port nearing completion since it means low-level consoles work, and the initial boot described above is

[1] The `btoc` macro converts bytes to clicks, which in FreeBSD are single pages. Mach allowed multiple pages to be managed as a virtual page.
[2] Most ports have one of these, but it seems to be misnamed, as it is not MIPS generic code.

```
Copyright (c) 1992-2013 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
        The Regents of the University of California. All rights reserved.
FreeBSD is a registered trademark of The FreeBSD Foundation.
```

Fig. 6 Copyright and Trademark Messages in FreeBSD 10.

complete. The MIPS port has some debugging output earlier in boot, but on mature platforms the copyright message is the first output from the kernel. Figure 6 shows the three messages printed at `SI_SUB_COPYRIGHT`.

The next sysinit of interest to porters is `SI_SUB_VM`. The MIPS `bus_dma(9)` implementation starts with a set of statically allocated maps to allow it to be used early in boot. The function `mips_dmamap_freelist_init` adds the static maps to the free list at `SI_SUB_VM`. The ARM platform does similar work, but does require `malloc` and thus runs `busdma_init` at `SI_SUB_KMEM` instead.

Further `bus_dma(9)` initialization takes place at `SI_SUB_LOCK` in the platform-specific, but often identical, `init_bounce_pages` function. It initializes some counters, lists, and the bounce-page lock.

All ports call a platform-specific `cpu_startup` function at `SI_SUB_CPU`, set up the kernel address space, and perform some initial buffer setup. Many ports also perform board-, SoC-, or CPU-specific setup such as initializing integrated USB controllers. Ports typically print details of physical and virtual memory, initialize the kernel virtual address space with `vm_ksubmap_init`, the VFS buffer system with `bufinit`, and the swap buffer list with `vm_pager_bufferinit`. On MIPS the platform-specific `cpu_init_interrupts` is also called to initialize interrupt counters.

Most platforms have their own `sf_buf_init` routine to allocate `sendfile(2)` buffers and initialize related locks. Most of these implementations are identical.

The bus hierarchy is established and device probing is performed at the `SI_SUB_CONFIGURE` stage (aka autoconfiguration). The platform-specific portions of this stage are the `configure_first` function called at `SI_ORDER_FIRST`, which attaches the nexus bus to the root of the device tree; `configure`, which runs at `SI_ORDER_THIRD` and calls `root_bus_configure` to probe and attach all devices; and `configure_final`, which runs at `SI_ORDER_ANY` to finish setting up the console with `cninit_finish`, and clear the `cold` flag. On MIPS and some other platforms `configure` also calls `intr_enable` to enable interrupts. A number of console drivers complete their setup with explicit sysinits at `SI_SUB_CONFIGURE`, and many subsystems such as CAM and `acpi(4)` perform their initialization there.

Each platform registers the binary types it supports at `SI_SUB_EXEC`. This primarily consists of registering the expected ELF header values. On a uniprocessor MIPS, this is the last platform-specific sysinit.

The final sysinit is an invocation of the `scheduler` function at `SI_SUB_RUN_SCHEDULER`, which attempts to swap in processes. Since `init(8)` was previously created by `create_init` at `SI_SUB_CREATE_INIT` and made runnable by `kick_init` at `SI_SUB_KTHREAD_INIT`, starting the scheduler results in entering userland.

Multiprocessor Support

Multiprocessor systems follow the same boot process as uniprocessor systems, with a few added sysinits to enable and start scheduling the other hardware threads. These threads are known as application processors (APs).

The first MP-specific sysinit is a call to `mp_setmaxid` at `SI_SUB_TUNABLES` to initialize the `mp_ncpus` and `mp_maxid` variables. The generic `mp_setmaxid` function calls the platform-specific `cpu_mp_setmaxid`. On MIPS, `cpu_mp_setmaxid` calls the architecturally-specific `platform_cpu_mask` to fill a `cpuset_t`

```
struct spin_entry {
        uint64_t        entry_addr;
        uint64_t        a0;
        uint32_t        rsvd1;
        uint32_t        pir;
        uint64_t        rsvd2;
};
```

Fig. 7 Definition of a `spin_entry` with Explicit Padding and the Argument Variables Renamed to Match MIPS Conventions.

with a mask of all available cores or threads. BERI's implementation extracts a list of cores from the DTB and verifies that they support the spin-table enable method. It further verifies that the spin-table entry is properly initialized or that the thread is ignored.

The initialization of APs is accomplished by the `mp_start` function called at `SI_SUB_CPU` after `cpu_startup`. If there are multiple CPUs, it calls the platform-specific `cpu_mp_start` and upon return prints some information about the CPUs. The MIPS implementation of `cpu_mp_start` iterates through the list of valid CPU IDs as reported by `platform_cpu_mask`, and attempts to start each one except itself as determined by `platform_processor_id`[3] with the platform-specific `start_ap`. The port-specific `platform_start_ap`'s job is to cause the AP to run the platform-specific `mpentry`. When run successfully, it increments the `mp_naps` variable, and `start_ap` waits up to five seconds per AP for this to happen before giving up on it.

Various mechanisms have been implemented to instruct a CPU to start running a particular piece of code. On BERI we have chosen to implement the spin-table method described in the ePAPR 1.0 specification [3] because it is extremely simple. The spin-table method requires that each AP have an associated `spin_entry` structure located somewhere in the address space and for that address to be recorded in the DTB. The BERI-specific definition of `struct spin_entry` can be found in Figure 7. At boot the `entry_addr` member of each AP is initialized to `1`, and the AP waits for the LSB to be set to `0`—at which time it jumps to the address loaded in `entry_addr` passing `a0` in register `a0`. We implement waiting for `entry_addr` to change with a loop in mini-boot. In BERI's `platform_cpu_mask` we look up the `spin_entry` associated with the requested AP, set the `pir` member to the CPU ID and then assign the address of `mpentry` to the `entry_addr` member.

The MIPS implementation of `mpentry` is assembly in `mips/mips/mpboot.S`. It disables interrupts, sets up a stack, and calls the port-specific `platform_init_ap` to set up the AP before entering the MIPS-specific `smp_init_secondary` to complete per-CPU setup and await the end of the boot process. A

3 Implemented in `mips/beri/beri_asm.S` on BERI.

ISILON The industry leader in Scale-Out Network Attached Storage (NAS)

Isilon is deeply invested in advancing FreeBSD performance and scalability. We are looking to hire and develop FreeBSD committers for kernel product development and to improve the Open Source Community.

We're Hiring!

With offices around the world, we likely have a job for you! Please visit our website at http://www.emc.com/careers or send direct inquiries to karl.augustine@isilon.com.

EMC² ISILON

typical MIPS implementation of **platform_init_ap** sets up interrupts on the AP and enables the clock and IPI interrupts. On BERI we defer IPI setup until after device probe, because our programmable interrupt controller (PIC) is configured as an ordinary device and thus cannot be configured until after **SI_SUB_CONFIGURE**.

The MIPS-specific **smp_init_secondary** function initializes the TLB, sets up the cache, and initializes per-CPU areas before incrementing **mp_naps** to let **start_ap** know that it has finished initialization. It then spins waiting for the flag **aps_ready** to be incremented, indicating that the boot CPU has reached **SI_SUB_SMP** as described below. On BERI it then calls **platform_init_secondary** to route IPIs to the AP and set up the IPI handler. The AP then sets its thread to the per-CPU idle thread, increments **smp_cpus**, announces itself on the console, and if it is the last AP to boot, sets **smp_started** to inform **release_aps** that all APs have booted, and sets the **smp_active** flag to inform a few subsystems that it is running with multiple CPUs. Unless it was the last AP to boot, it spins waiting for **smp_started** before starting per-CPU event timers and entering the scheduler.

The final platform-specific sysinit subsystem is **SI_SUB_SMP**, which platform-specific **release_aps** functions are called to enable IPIs on the boot CPU, inform previously initialized APs that they can start operating, and spin until they do so as described above. In the MIPS case this means atomically setting the **aps_ready** flag to **1** and spinning until **smp_started** is non-zero.

A Word on IPIs

In multiprocessor (MP) systems, CPUs signal each other via Inter-Processor Interrupts (IPIs). Various IPI mechanisms exist, with FreeBSD MIPS using the simplest model, a per-CPU integer bitmask of pending IPIs and a port-specific mechanism for sending an interrupt—almost always to hardware interrupt 4. This is implemented by the **ipi_send**, which is used by the public **ips_all_but_self**, **ipi_selected**, and **ipi_cpu** functions. MIPS IPIs are handled by **mips_ipi_handler**, which clears the interrupt with a call to **platform_ipi_clear**, reads the set of pending IPIs, and handles each of them.

On BERI IPIs are implemented using the BERI PIC's soft interrupt sources. IPIs are routed by **beripic_setup_ipi**, sent by **beripic_send_ipi**, and cleared by **beripic_clear_ipi**. These functions are accessed via **kobj(9)** through the FDT_IC interface defined in **dev/fdt/fdt_ic_if.m**. The internals of BERI PIC are described in the BERI Hardware Reference [4].

Acknowledgments

We thank our colleagues—especially Jonathan Anderson, Ruslan Bukin, David Chisnall, Nirav Dave, Alexandre Joannou, Wojciech Koszek, Ben Laurie, A. Theodore Markettos, Ed Maste, Simon

REFERENCES

[1] Heinrich, J. *MIPS R4000 Microprocessor User's Manual, Second Edition*. (1994)

[2] Markettos, A. T.; Woodruff, J.; Watson, R. N. M.; Zeeb, B. A.; Davis, B.; and Moore, S. W. *The BERIpad tablet: open-source construction, CPU, OS and applications* (http://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2013terasic-beri-submitted.pdf), Proceedings of 2013 FPGA Workshop and Design Contest, Southeast University, Nanjing, China. (November 1–3, 2013) [3] Power.org, Power.org Standard for Embedded Power Architecture Platform Requirements (ePAPR). (2008)

[4] Watson, R. N. M.; Woodruff, J.; Chisnall, D.; Davis, B.; Koszek, W.; Markettos, A. T.; Moore, S. W.; Murdoch, S. J.; Neumann, P. G.; Norton, R.; and Roe, M. *Bluespec Extensible RISC Implementation: BERI Hardware Reference* (http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-852.pdf) , Technical Report UCAM-CL-TR-852, University of Cambridge, Computer Laboratory. (April 2014)

[5] Watson, R. N. M.; Chisnall, D.; Davis, B.; Koszek, W.; Moore, S. W.; Murdoch, S. J.; Neumann, P. G.; and Woodruff, J. *Bluespec Extensible RISC Implementation: BERI Software Reference* (http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-853.pdf) , Technical Report UCAM-CL-TR-853, University of Cambridge, Computer Laboratory. (April 2014)

[6] Woodruff, J.; Watson, R. N. M.; Chisnall, D.; Moore, S. W.; Anderson, J.; Davis, B.; Laurie, B.; Neumann, P. G.; Norton, R.; and Roe, M. "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *Proceedings of the 41st International Symposium on Computer Architecture* (ISCA 2014). (June 2014)

[7] McKusick, M. K.; Neville-Neil, G. V.; Watson, R. N. M. *The Design and Implementation of the FreeBSD Operating System, Second Edition*, Boston, Massachusetts: Pearson Education. (September 2014)

W. Moore, Alan Mujumdar, Steven J. Murdoch, Peter G. Neumann, Philip Paeps, Michael Roe, Colin Rothwell, Hans Petter Selasky, Stacey Son, and Bjoern A. Zeeb.

This paper is approved for public release; distribution is unlimited. It was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under Contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. •

BROOKS DAVIS is a Senior Software Engineer in the Computer Science Laboratory at SRI International and a Visiting Research Fellow at the University of Cambridge Computer Laboratory. He has been a FreeBSD user since 1994, a FreeBSD committer since 2001, and was a core team member from 2006 to 2012. His computing interests include security, operating systems, networking, high performance computing, and, of course, finding ways to use FreeBSD in all these areas.

ROBERT NORTON is a PhD student at the University of Cambridge. He is working on the CHERI CPU, a project to improve application security using hardware support for fine-grained memory protection and compartmentalization. He is particularly focusing on aspects of sandboxing and transitions between security domains.

JONATHAN WOODRUFF is a researcher at the University of Cambridge. He is a core developer of the BERI/CHERI open-source CPU and has helped develop the CHERI ISA for memory safety and isolation for practical hardware implementation. He has also worked on large processor architecture simulation on FPGA. He supports open-source processor research enabled by FPGAs for reproducible, full-system research that reaches to customized hardware.

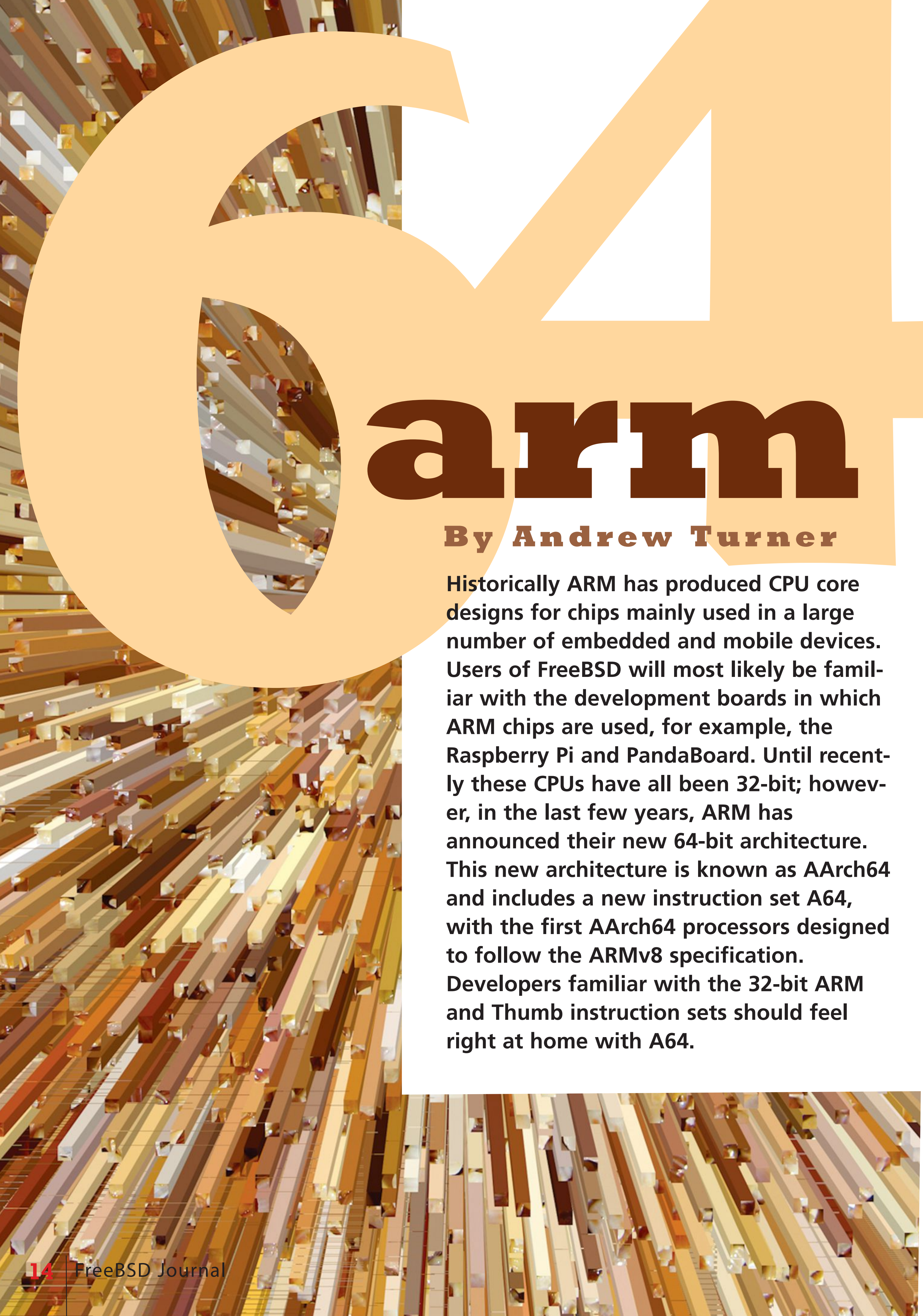
DR. ROBERT N. M. WATSON is a University Lecturer in Systems, Security, and Architecture at the University of Cambridge Computer Laboratory; FreeBSD developer and core team member; and member of the FreeBSD Foundation board of directors. He leads a number of cross-layer research projects spanning computer architecture, compilers, program analysis, program transformation, operating systems, networking, and security. Recent work includes the Capsicum security model, MAC Framework used for sandboxing in systems such as Junos and Apple iOS, and multithreading in the FreeBSD network stack. He is a coauthor of *The Design and Implementation of the FreeBSD Operating Systems (Second Edition)*.

RootBSD

Premier VPS Hosting

RootBSD has multiple datacenter locations, and offers friendly, knowledgeable support staff. Starting at just $20/mo you are granted access to the latest FreeBSD, full Root Access, and Private Cloud options.

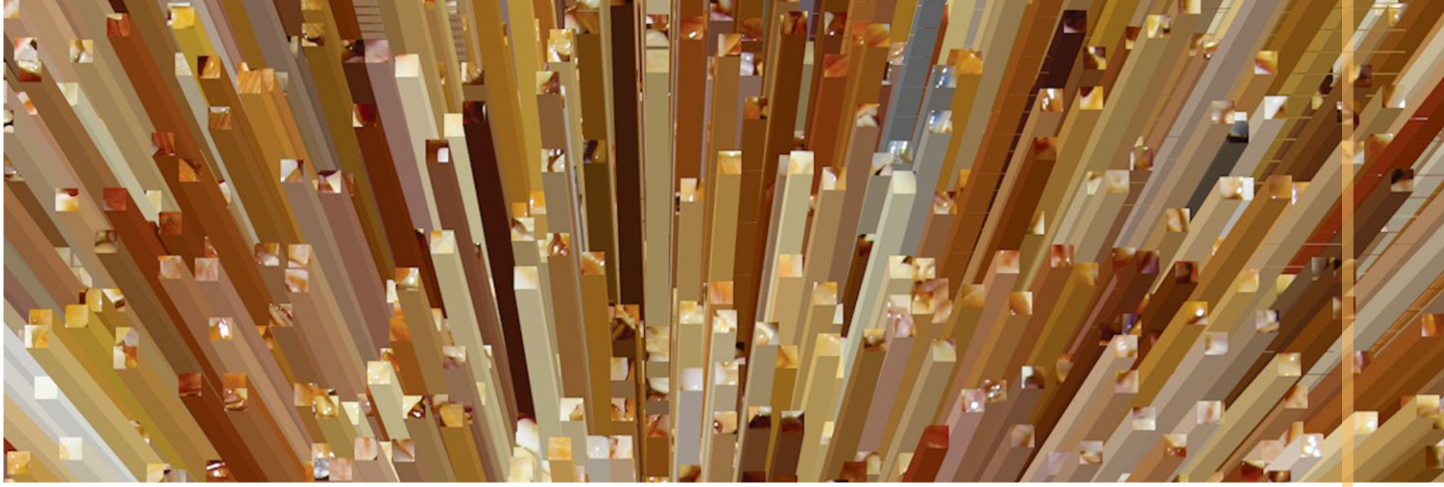
www.rootbsd.net



64 arm

By Andrew Turner

Historically ARM has produced CPU core designs for chips mainly used in a large number of embedded and mobile devices. Users of FreeBSD will most likely be familiar with the development boards in which ARM chips are used, for example, the Raspberry Pi and PandaBoard. Until recently these CPUs have all been 32-bit; however, in the last few years, ARM has announced their new 64-bit architecture. This new architecture is known as AArch64 and includes a new instruction set A64, with the first AArch64 processors designed to follow the ARMv8 specification. Developers familiar with the 32-bit ARM and Thumb instruction sets should feel right at home with A64.



The ARMv8 Specification

The first change developers will notice is the increase in size and large number of the general purpose registers—there are 31, 64-bit, general-purpose registers. In addition, the program counter and stack pointer are no longer part of the general purpose register set and can only be accessed through a few instructions.

Another large change is removing the ability for most instructions to be conditionally executed. Now, only the branch instruction is able to be conditionally executed. This conditional execution was one of the features of earlier chips; however, they used 4 bits from each instruction, leaving only 28 bits to encode the instruction.

The ARM naming scheme can be confusing. ARMv8, in this case, describes the instruction sets available along with other important architectural areas like how the cache is expected to work, or how to program the MMU. The architecture will be backwards compatible with the previous revision, but may add a new flag or an incompatible option. An example of the latter is that some ARMv7 designs have added support for larger physical addresses in the MMU, known as the Large Physical Address Extension, or LPAE. With these, the existing page table format will still work, but the operating system can use the new format to access more than 4GB of physical memory.

Not only are there ARMv names, but there are also design families. For ARMv7, ARM has its Cortex-A designs. These designs are all ARMv7 designs, but have some different characteristics—e.g., they may have different pipeline lengths or may implement LPAE support. There is also an option for a company to design its own ARM core, known as an Architecture License, where the licensees design their own compatible core. One company that has taken this route is Marvell, which implements an ARMv7 compatible core of their own design. With ARMv8, ARM has endeavored to have a larger number of independent designs.

The last level is the implementation of the core—for example, the Cortex-A7. This is an ARMv7 design with LPAE, and, from the software point-of-view, is architecturally identical to the Cortex-A15. This is important, for it allows pairs of cores to be used together in an SMP system—for instance, having four power-efficient Cortex-A7 cores and two high-performance Cortex-A15 cores. The software can turn these on and off as needed. This configuration is known as big.LITTLE, and is designed to provide improved battery life while offering performance when required.

The ARMv8 architecture follows this by introducing the Cortex-A53 and Cortex-A57 cores from ARM as well as a number of third-party designs. One example is Project Denver from Nvidia, a design that takes the ARM instructions and translates them into an internal instruction set that may, at a later stage, be optimized. This is a continuation of the Transmeta x86 chips, just with a new input instruction set.

ARMv8 also changes the operation of exception states. These exception states can be thought of as different privilege levels. These were rather complex in the 32-bit chips, where interrupt and system-call handling are different states and where they both have the same privilege. With ARMv8 there are four exception levels; the lower three can be thought of as the userland, kernel, and hypervisor states. The most privileged level is normally programmed by the chip vendor and is used to abstract some of the chip's functionality, for example, to provide a power-management interface. These states are known as EL0 to EL3 where the larger number is at higher privilege. These could be used as EL0 for userland, EL1 for the kernel, and EL2 for bhyve of xen.

The arm64 Project

I have been working on this project in my spare time for the last two years, and, as of November 2014, as part of a FreeBSD Foundation project to port FreeBSD to the AArch64. Initially, this involved writing simple code to bootstrap the hardware and to allow jumping to C code. Later, this was brought into the FreeBSD tree in a project branch. Finally, for the FreeBSD Foundation project, this has moved to GitHub to allow collaboration among a number of developers. This new architecture will be known in FreeBSD as arm64.

The project has two goals: The first, and where I am focusing my time, is to lay the foundation for FreeBSD to run on a number of different chips, which includes the internal kernel infrastructure and userland. There is a second goal, which is to have FreeBSD run on the Cavium ThunderX processor. This is a server chip with up to 48 ARMv8 cores that can operate in a dual-chip configuration allowing for 96 cores.

Working on porting an operating system to a new platform before hardware is available can be difficult. Luckily, ARM released a version of their simulator—known as the Foundation Model—so that developers could test their code. This has been where all of my work has taken place, including getting to the point where userland can be run.

Most of the code written for any new port is within the machine-dependent parts of the kernel, and the arm64 port is no exception. There are a number of steps required to get a new port with the first being a toolchain. When the project was started, there was only a single toolchain available—gcc with binutils—which required me to port both to build FreeBSD binaries. (This was mostly copying the existing Linux and modifying as needed.) Having previously worked on the ARM EABI, I realized this required a similar approach, and so I was familiar enough with both projects to get them working.

But a toolchain is just the first step. One also needs either hardware or a simulator to test booting the hardware. This is where the Foundation Model was useful, as it allowed for experimentation a number of years before the hardware would arrive. And ARM released a simple boot wrapper to initialize the simulator to be used by Linux. As this is under a BSD license, there were no issues involved in modifying it to be used as a starting point for experimentation.

Initially, these experimentations were simple, as there was no documentation—just an overview of the instruction set. But, as is too often the case with FreeBSD, this required inferring how the hardware worked by looking at Linux and finding the similarities with ARMv7. Some things—for example, the MMU page tables—are very similar and can be worked on, whereas others, like the system registers, have been changed and so they may or may not be the same. Even without this documentation, I progressed to the point where I could move from pure assembly to executing C code. The MMU was still turned off, but this allowed the rest of the code to be tested.

In September 2013, ARM released the ARMv8 Architecture Reference Manual, known as the ARM ARM. This allowed the last parts to be written for enabling the MMU and for any magic numbers to be correctly documented. With this, the code could be moved from GitHub into a FreeBSD project branch. As part of this, llvm and clang in the base could be used, as they are new enough to generate AArch64 code, and with very little work can generate code for FreeBSD. A copy of binutils is still needed to provide a few missing FreeBSD tools, as the copy in base is too old.

The Boot Environment

It is expected that the boot environment on AArch64 servers will be UEFI based, and, to simplify booting, having a UEFI-enabled FreeBSD loader is an important step in the process. When working on the port, the loader can be thought of as a simple, single-threaded kernel. Unfortunately, the UEFI boot environment needs the loader to be an EFI binary. The process is a little more difficult with AArch64 than with other UEFI platforms, as the AArch64 binutils is unable to convert from an ELF file to the needed PE+. To work around this, a combination of assembly, linker scripts, and the use of objcopy is needed to create the image. A PE+ header is created in an assembly file, a linker script provides the needed information as to the size of the binary, and objcopy will copy from the ELF image to a binary image, leaving the PE+ header at the beginning of the binary.

Even with the loader being in the correct format, more work is required for it to run and load the kernel. First the loader may need to be relocated in a similar process to loading a dynamic library. A few functions also need to be implemented, including functions to copy data between physical memory and what will be the kernel's virtual memory. Most of these, along with the main function, can be copied and adjusted as needed from an existing port.

The Kernel

For AArch64 I decided to use the semihosting file system to load the loader and kernel. Semihosting is a method where software running on an ARM processor may access the host environment. With physical hardware, this works through the debug adapter, while the simulator allows for any file to be accessed in the directory from which it was run. The UEFI environment exports this as a file system available to any EFI applications. To access this, a new file system handler was written that included working around bugs in the UEFI implementation. With this, the kernel can be loaded without creating a disk image, thus allowing faster development.

Having gotten the loader to the point where it was capable of running and loading a kernel, the next step was to build the kernel for it to load and run. The kernel needs a number of machine-dependent functions to be written. To begin with, these can be stub functions where each called panic printing its name. With the kernel building, it needs to be loaded and have initial code to run. As I had written my earlier experimentations with a plan to import them into FreeBSD, this simplified the process, as I already had code to start execution of the kernel.

But even with this code in the kernel, we are still only able to get into the kernel's C code. From here, work is needed to continue the early bootstrap. This includes parsing any kernel metadata passed to it by the loader, bootstrapping pmap, and initializing the kernel. The kernel metadata included data describing the hardware, which could be either a Flattened Device Tree Blob, or ACPI data. By having the loader pass in this data, we are able to keep the kernel generic.

FreeBSD has a layer named pmap to handle the machine dependent parts of the virtual memory subsystem, for example, updating the MMU to adjust which physical address a virtual address points to. This will be different for each architecture, and, as is the case with 32-bit ARM and PowerPC, may need different implementations within a single architecture. A decision was made to include a single, large, direct mapped region on arm64. This simplifies the code when a physical address to virtual address calculation is needed. Bootstrapping the pmap layer can be a little tricky. The existing mappings need to be taken into account and new mappings added.

Working with the page tables can often be problematic, as configuring an entry without performing the correct TLB invalidation can lead to unexpected results. To assist with this, ARM has added a useful instruction to see how the hardware is performing the address translation. This instruction, the AT instruction, performs an address translation and reports back the result for the developer to inspect. This, along with software-based page table walking, has proved to be useful in tracking down problems. The address translation instruction can report back a number of useful bits of information, including whether or not the translation was successful and either the physical address or a flag to describe why it failed. It is also able to attempt the translation on both userland and kernel addresses. It is common for this instruction to be the first place to check when the kernel exhibits odd behavior, as was the case recently where the shell was crashing just after forking to execute a command. With the help of this feature, the problems with the context switching and pmap code were found and fixed.

Having finished this early platform initialization, the kernel enters into the machine independent code. Yet even at this stage it will still call back into the machine dependent code. One place the kernel performs this is for device enumeration. On FreeBSD we have a tree of devices, the root of which is the nexus device. The job of nexus is to handle any bus drivers the architecture may need, and for this it needs to implement a number of resource handling functions. These involve allocating and releasing device memory and interrupts, and configuring the interrupt. This configuration is needed, as there may be a few different interrupt controllers—which is the case on 32-bit ARM. Depending on the configuration, there may be the standard ARM Generic Interrupt Controller; however, this is only common on chips that support SMP. On older designs, designers used their own interrupt controllers.

Nexus also needs to handle device memory allocation. For this it configures the base case for memory resource in the resource management abstraction. The driver needs to handle any physical addresses that are to be allocated from. FreeBSD will then restrict the size of the physical memory to manage the use of Open Firmware and Flattened Device Tree buses. It is expected that ACPI will work in a similar way; however, work has yet to start on this, as most of the ARMv8 focus thus far has been to use Flattened Device Tree, even on Linux.

Having nexus and the Open Firmware and Flattened Device Tree devices is not enough to boot a system. We also need, at a minimum, an interrupt controller, a timer, and some form of console. On the Foundation Model, these are the Generic Interrupt Controller version 2, the Global Timer, and an ARM pl011 UART, respectively. All of these are already supported by FreeBSD; however, work was needed for them to work on ARMv8. The simplest was the UART. Due to how ARM specified the device tree for the Foundation Model, the base address of the parent bus needed to be hard-coded, but this was later fixed to correctly parse the data.

The interrupt controller and timer drivers needed more work. The interrupt controller was changed to allow support for multiple controllers. This is based on the PowerPC design, but it is expected this will be replaced with the updated ARM design when this separate project is integrated into CURRENT. With the timer, the method for accessing timer values on 32-bit ARM is through a system co-processor. This is accessed using instructions for moving data between this co-processor and ARM registers. On AArch64 these have moved to special registers. Along with this, there are two groups of these registers, one for the physical timer and another for the virtual timer. This virtual timer is set up to be the physical timer plus a random delta. Normally the kernel will only have access to the virtual timer; however, it may get access to the virtual timer in some cases, for example, as a Hypervisor. To handle this, the driver's internal API was updated to allow access to either of these modes depending on the situation.

The kernel won't get very far with just these three devices. The first problem is that it needs to read and write to device memory. This abstraction is known as bus space and allows a single driver to talk to a bus without knowing the details of the bus. Bus space provides a number of functions to read and write to a device, and, eventually, all of these need to be written. During the early work only a few are needed, the main group being the reading and writing of a single item, be it 1, 2, 4 or 8 bytes. Without this, no drivers are able to access the devices they are controlling.

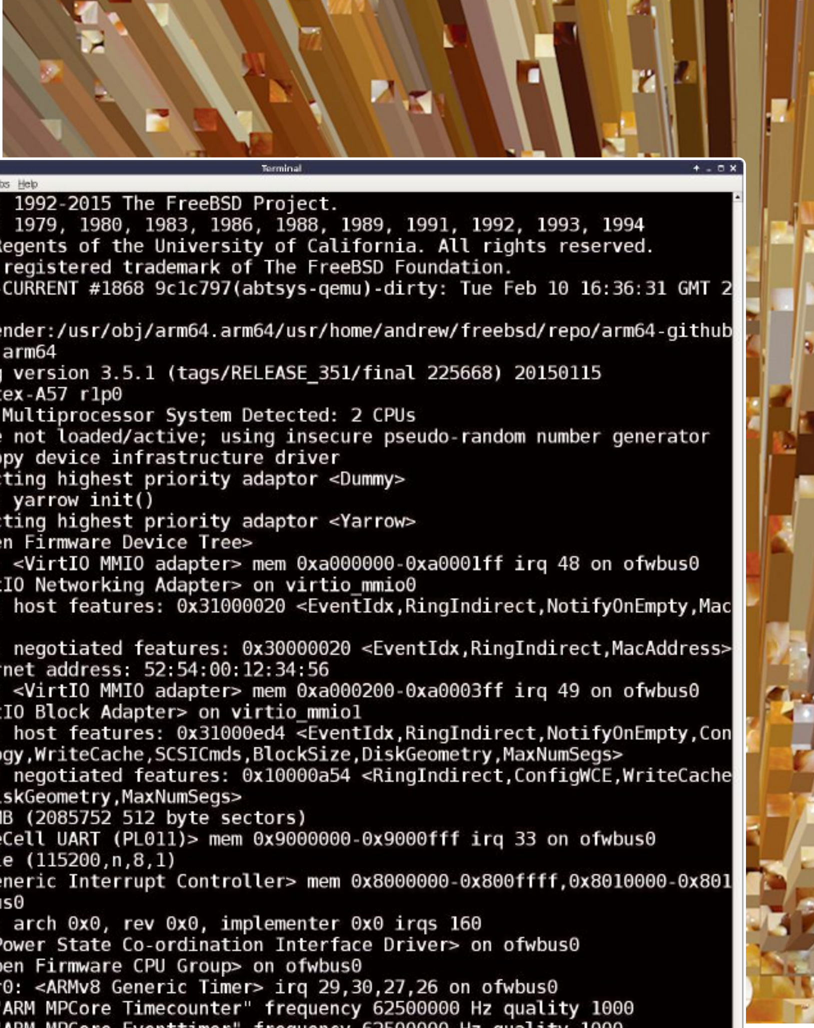
The bus space abstraction also handles any mapping and unmapping of these bus resources. As the internal ARM device bus is memory mapped, this requires bus space to allocate a block of virtual addresses and then map them to the correct physical address. For this, the existing 32-bit ARM code can be reused with minimal changes.

There are a few other parts of the kernel that need to be implemented for early booting. The kernel expects to create kernel threads. For this, the kernel has a few functions to configure or needs to duplicate new threads and processes and to switch between them. Some of these need to duplicate an existing process's data or create it from scratch, whereas others handle writing the state of the current process to memory and loading a new process's state along with any cache handling requirements. It can be tricky to get these correct and one good way to test them is to get the userland working, as it will quickly show when, for example, one process is accessing the wrong memory and changing another process's stack.

It is unlikely to get to this point without hitting a hardware exception. The main cause will be some sort of memory fault—for example, when the kernel is accessing an invalid virtual address. To handle this, I wrote a simple exception handler. It needed to store the register state, call the appropriate handling function, then restore the registers and return to where it was executing from. Early on, the handler could be simple, as it only needed to handle the kernel map where the address would have the top bit set. If we found an invalid address, the kernel could dump the saved registers to help track down the issue.

At this stage, configuring interrupts worked along with allocating and accessing device memory, process creation and switching, and simple exception handling. This allowed the kernel to reach its first major milestone, getting to the mountroot prompt. This is the point where enough of the system is running that the kernel tries to mount the root file system and fails. The scheduler is running off the timer interrupt, the system UART is working, and any triggered exceptions are at least handled, even if it is to panic.

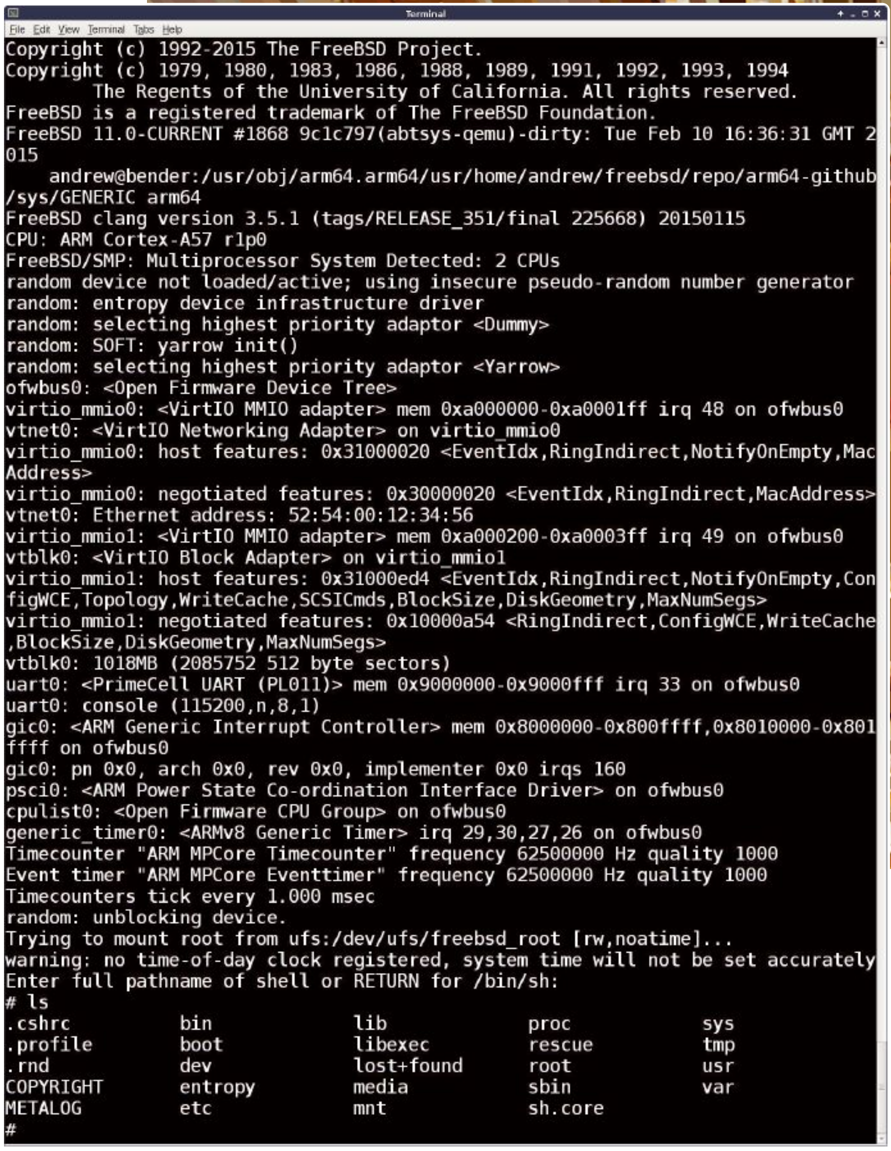
Having gotten this far with the project, there is just a kernel running. The userland is still not working, and it needs some sort of file system. FreeBSD has the ability to embed a small file system image into the kernel, but to build this, bits of userland need to be ported, and the system call interface needs to be defined.



AArch64 handles system calls in the kernel as hardware exceptions, with the kernel deciding how to decode which system call userland is accessing. For this, I have decided to follow the example of Linux and store the exception value in a register. Both libc and the kernel need to be updated to encode or handle these respectively. Along with this libc, functions such as setjmp and longjmp need to be written. Initially, only a static library is needed, which can simplify the early work. As more of userland is ported, the required libc functions can be added. On arm64, only a very minimal set of functions has been implemented, but more are being worked on as needed.

To start executing an application, there is a small piece of machine-dependent userland code named csu. This has to load argc and argv from memory along with the environment and a few other pointers set by the kernel. There may also be code that needs to be run before main, for example, to create static C++ objects. This is all handled by csu.

Most of the userland libraries needed to get a simple file system can be built with this. The minimum requirements for a usable file system are init and sh. Having built both of these, I used the makefs command. This takes a directory structure and generates a UFS file system image. If the size of this image is small enough, it can be embedded into the kernel; a 4-MB image is large enough to hold a static and stripped init, sh and ls.



```
Copyright (c) 1992-2015 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
        The Regents of the University of California. All rights reserved.
FreeBSD is a registered trademark of The FreeBSD Foundation.
FreeBSD 11.0-CURRENT #1868 9c1c797(abtsys-qemu)-dirty: Tue Feb 10 16:36:31 GMT 2
015
    andrew@bender:/usr/obj/arm64.arm64/usr/home/andrew/freebsd/repo/arm64-github
/sys/GENERIC arm64
FreeBSD clang version 3.5.1 (tags/RELEASE_351/final 225668) 20150115
CPU: ARM Cortex-A57 r1p0
FreeBSD/SMP: Multiprocessor System Detected: 2 CPUs
random device not loaded/active; using insecure pseudo-random number generator
random: entropy device infrastructure driver
random: selecting highest priority adaptor <Dummy>
random: SOFT: yarrow init()
random: selecting highest priority adaptor <Yarrow>
ofwbus0: <Open Firmware Device Tree>
virtio_mmio0: <VirtIO MMIO adapter> mem 0xa000000-0xa0001ff irq 48 on ofwbus0
vtnet0: <VirtIO Networking Adapter> on virtio_mmio0
virtio_mmio0: host features: 0x31000020 <EventIdx,RingIndirect,NotifyOnEmpty,Mac
Address>
virtio_mmio0: negotiated features: 0x30000020 <EventIdx,RingIndirect,MacAddress>
vtnet0: Ethernet address: 52:54:00:12:34:56
virtio_mmio1: <VirtIO MMIO adapter> mem 0xa000200-0xa0003ff irq 49 on ofwbus0
vtblk0: <VirtIO Block Adapter> on virtio_mmio1
virtio_mmio1: host features: 0x31000ed4 <EventIdx,RingIndirect,NotifyOnEmpty,Con
figWCE,Topology,WriteCache,SCSICmds,BlockSize,DiskGeometry,MaxNumSegs>
virtio_mmio1: negotiated features: 0x10000a54 <RingIndirect,ConfigWCE,WriteCache
,BlockSize,DiskGeometry,MaxNumSegs>
vtblk0: 1018MB (2085752 512 byte sectors)
uart0: <PrimeCell UART (PL011)> mem 0x9000000-0x9000fff irq 33 on ofwbus0
uart0: console (115200,n,8,1)
gic0: <ARM Generic Interrupt Controller> mem 0x8000000-0x800ffff,0x8010000-0x801
ffff on ofwbus0
gic0: pn 0x0, arch 0x0, rev 0x0, implementer 0x0 irqs 160
psci0: <ARM Power State Co-ordination Interface Driver> on ofwbus0
cpulist0: <Open Firmware CPU Group> on ofwbus0
generic_timer0: <ARMv8 Generic Timer> irq 29,30,27,26 on ofwbus0
Timecounter "ARM MPCore Timecounter" frequency 62500000 Hz quality 1000
Event timer "ARM MPCore Eventtimer" frequency 62500000 Hz quality 1000
Timecounters tick every 1.000 msec
random: unblocking device.
Trying to mount root from ufs:/dev/ufs/freebsd_root [rw,noatime]...
warning: no time-of-day clock registered, system time will not be set accurately
Enter full pathname of shell or RETURN for /bin/sh:
# ls
.cshrc          bin             lib             proc            sys
.profile        boot            libexec         rescue          tmp
.rnd            dev             lost+found      root            usr
COPYRIGHT       entropy         media           sbin            var
METALOG         etc             mnt             sh.core
#
```

Having implemented all the required kernel stub functions, and with a little luck, init should start executing. To begin with, only indirect evidence will exist, for example, by observing the state in exception handlers. As more bugs are fixed, init should get further along until if init is set up to boot to single-user mode, it will print out a message to select the shell. There may be further fixes required at this point, but the kernel should be mostly ready for userland, and init will try to run the shell, which can be used to run any other applications in the file system.

This is the point to which the arm64 port has progressed as of the time of this writing. FreeBSD can boot with a kernel file system and run static binaries on the Foundation Model. Work has started on loading dynamic executables; however, this is still only running from single user mode.

Further work is progressing toward a stable port. In the short term, work is being done to get FreeBSD running on the Cavium ThunderX hardware, along with support for dynamic binaries and multi-user mode. The port is still only running on a single core, and getting SMP working is a requirement due to the large number of cores in the ThunderX.

Looking further ahead, there will need to be stability work to make sure the code will work when used in a production environment. One way to perform this is to attempt to build ports on the hardware. This will also give us a chance to see the state of the ports tree on this new platform. There is also interest in other items like

DTrace and hwpmc. Along with this there will always be work to port FreeBSD to any new hardware that turns up, as, unfortunately, a large number of devices are specific to each hardware vendor.

Acknowledgments

I would like to thank the FreeBSD Foundation, along with ARM Ltd and Cavium for sponsoring the project. As this is just the start, further work will be needed. The chip vendors need to release their documentation so FreeBSD can be ported to their chips. Further sponsorship will also help in getting FreeBSD to a stable and production-ready state on ARMv8. •

Andrew Turner started with FreeBSD on ARM by porting it to the Samsung CPU within the OpenMoko phones and is responsible for bringing in ARM EABI support to FreeBSD. He has worked as an embedded software engineer on projects from deeply embedded ARM devices with a few kB of RAM, to multi-core ARM boards with many gabytes of memory. He also works as a contractor porting FreeBSD to ARMv8 chips.

FURTHER READING

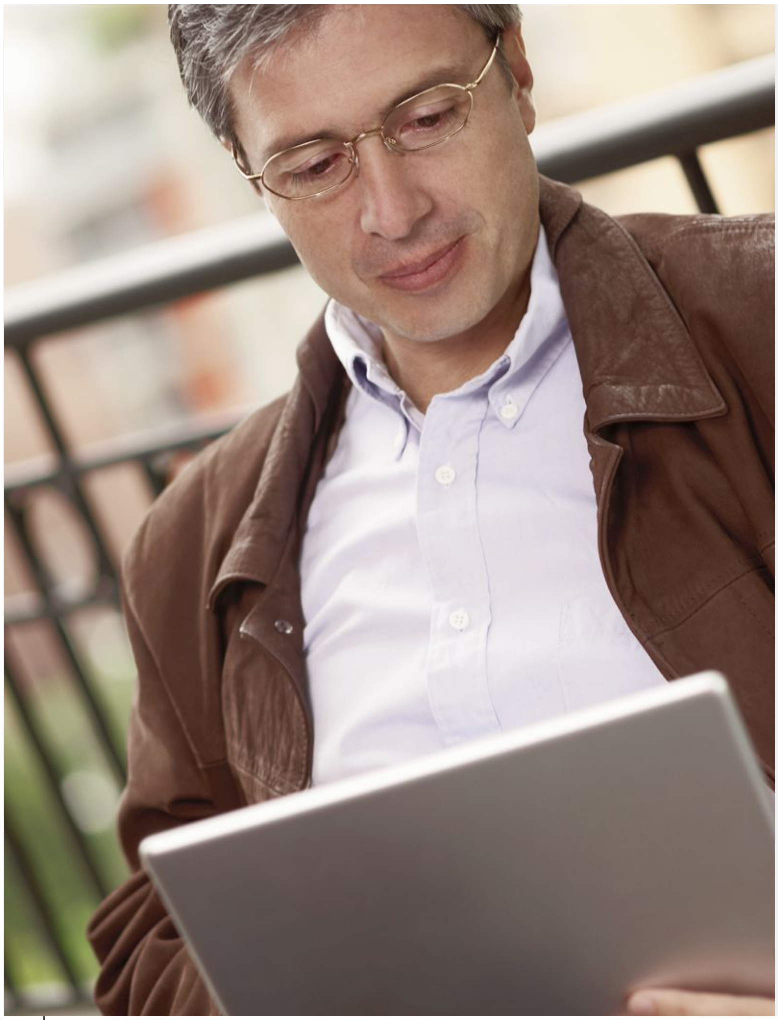
The FreeBSD wiki – https://wiki.freebsd.org/arm64
Technology Preview: The ARMv8 Architecture – http://www.arm.com/files/downloads/ARMv8_white_paper_v5.pdf
ARM Architecture Reference Manual –
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html

INTERACTING with the FreeBSD Project

by Dru Lavigne

Getting Involved

As an open-source project, FreeBSD relies on the ongoing contributions and diverse skills of a vibrant community in order to create and maintain software that meets the needs of its users. Perhaps you have thought about contributing to the FreeBSD Project but are not sure how to get started. Or, perhaps you already contribute to the Project, but would like to encourage others to get involved as well.



This article provides a general overview of how and why to start contributing to the FreeBSD Project. It also addresses some of the concerns and questions commonly voiced by new contributors.

But I Don't Write Code!

It's true: the FreeBSD Project creates an operating system that is primarily written in the C programming language. While developers are always needed, this does not mean that the only contributions that the Project accepts are in the form of C code. Examples of non-code contributions include, but are not limited to:

Documentation: The FreeBSD Project provides man pages, which include working examples, and a comprehensive Handbook for using FreeBSD. This documentation is an ever-changing target as new features are added to software releases and users come up with new use cases for using the software. It takes the contributions of writers, who create new documentation; editors, who review and fine-tune documentation changes; and users, who report outdated or incorrect documentation, to keep the documentation resources useful and up-to-date.

Translations: FreeBSD is an international community and English is not the first language for many users. Having documentation available in one's native language significantly reduces the learning curve for new users. If English is not your first language, consider assisting with the translation of the Handbook into your native language and the impact that translation can have for users in your geographic region.

Events and News Items: The main page of the FreeBSD website is typically one's first exposure to FreeBSD. Submitting news items and upcoming events is a quick and effective way to make sure this content reflects the dynamic and far-reaching nature of the Project.

Ported Applications: The extensive collection of ports, or software that has been tested to run on FreeBSD, is one of the compelling features of using FreeBSD. At over 24,400 software applications and growing, it's clear that it takes the contributions of many users to test and keep applications up-to-date as new versions become available. Maintaining an existing port or creating a new port for an application that currently isn't available for FreeBSD helps to keep the software collection available to other users while also providing a way to improve your own development or scripting skills.

Testimonials and Case Studies: Does your company use FreeBSD or do you use FreeBSD to make your livelihood? The Project is always interested in showcasing where, how, and why FreeBSD is used.

User Support: The FreeBSD Project provides many resources so that members of the community can assist each other, including forums, mailing lists, and IRC channels. Taking the time to answer unanswered questions that you know the answer to or to point new users to the resources they need may not seem like much, but it helps to build a strong community.

Social Media: Do you live on Twitter? Consider adding Tweets of interest to FreeBSD users using #freebsd. Are you doing interesting things with FreeBSD or trying out new things as you learn to use FreeBSD? Consider blogging your efforts, as they will be of interest to other FreeBSD users while adding to the collective knowledge of "yes, you can do xyz on FreeBSD and here's how." Or, if you prefer to talk instead of write, contact the folks at bsdnow.tv or bsdtalk.org to arrange for a short interview.

Monetary Support: Is life just too busy for you right now to contribute time, but you still want to assist the Project? Consider making a monetary donation to the FreeBSD Foundation, knowing that the funds will be used to assist the Project. Before doing so, check to see if your employer has a donation matching program or if they would be interested in becoming a financial sponsor of the Foundation.

Advocacy: Do you attend any technical conferences or have a local technical user group? Consider giving a presentation, lightning talk, or BoF (Birds of a Feather) session about what you are doing with FreeBSD. If your presentation is accepted, send a note to the freebsd-doc mailing list so that the information about the event can be listed on the FreeBSD website.

Development: While this section has concentrated on non-code contributions, it is worth noting that contributions from developers are still needed. As an entire operating system, FreeBSD offers many coding opportunities: device drivers, networking protocols, file systems, APIs, and hardware subsystems, to name a few.

But I'm Not an Expert!

Hopefully, some of the contribution areas caught your interest. But you may be thinking "I'm still very new at FreeBSD and have so much to learn! Shouldn't I wait until I'm an expert before I start contributing?" In short, the answer is "no, don't wait." You can start contributing using what you know now and the type and depth of your contributions can grow as you gain experience.

The FreeBSD Project does have a lot of contributors who are very good at what they do and who are known as experts in their field. This can be pretty intimidating to new or casual users who are interested in contributing, but fear that their contributions are too small or too basic to make any difference. If you feel this way, don't forget that the community is much, much larger than the names that you are familiar with and that a large portion of the community is learning as they go and relies on the contributions of other users.

Why Get Involved?

This is a very good question as it sometimes seems that contributing to an open-source project is a one-way street. Sure, it is nice to give back when you're using something for free, but we all have only so much time and energy to spare. In reality, there are many compelling (and even downright self-serving) reasons for contributing back to any open-source project. The actual reasons will vary depending upon the people and their goals, but here are some examples that apply to contributing to open source in general:

Students, the unemployed or underemployed, and those individuals looking to improve their job prospects need a way to differentiate themselves in the job market. Open-source contributions, especially over time, can be used to show initiative

(I did this in my spare time), to learn from senior-level technologists, to improve technical skills (here is an online list of my contributions), to improve soft skills (I've worked with others and within processes as part of a large, organized project), and to cultivate contacts with experts in the field.

It's fun! Where else do you get to interact with people from all over the world who happen to share your passion for that bit of the open-source world that has caught your interest? This is especially true if you are the only person in your family, workplace, or town that has even heard of that bit of open-source technology. It can be a relief to work with others without having to first figure out which terms to use in order to explain what it is you do.

It's free! Where else can you learn about the internal components of an operating system and how these evolve over time, with nothing more than an Internet connection and your spare time? FreeBSD is particularly well suited for this purpose. Not only is it well documented, its code and commit history have been publicly available since the Project's inception in 1993.

There's no glass ceiling. Since open-source contributions come in over the Internet, no one needs to know your age, gender, race, economic status, or sexual orientation. Contributors become known by the quality of their contributions and the areas in which they contribute. Stick around for a while and you'll find that you have turned into a respected expert in your niche.

Organizations that build a product or service using an open-source technology quickly learn to appreciate the pain that comes with maintaining their own fork of that project. Even if contributing back patches and improvements is not required by the open-source license, the amount of development resources that can be saved when doing so quickly adds up. In addition, organizations that are looking for a competitive advantage do well to cultivate a working relationship with the project they use: Being an active participant increases the likelihood that your voice is heard when the Project is considering new features and its development roadmap.

Why FreeBSD?

The previous section outlined some advantages to contributing to open source in general. Why contribute to FreeBSD in particular, given that there are literally thousands of open-source projects looking for contributors?

Before contributing to any open-source project, it is worthwhile to research what that project excels at to make sure that it matches your interests and goals. FreeBSD excels in many technical areas, including:

Networking: FreeBSD inherited an excellent networking heritage from the original Berkeley Software Distribution and continues to innovate in the networking field. For example, did you know that BSD sockets provided the original API for TCP/IP? Or that FreeBSD provided the first IPv6-only networking stack? Recent innovations include being the research platform for modular TCP congestion control algorithms and the development of netmap for testing high performance direct-to-hardware packet I/O.

Security: FreeBSD is well known for being a secure operating system and for working closely with security researchers as a reference platform for new security frameworks. Jails have provided lightweight isolation and operating system virtualization since 2000 and continue to see improvements such as fine-tuned resource limits. The Capsicum reference platform is defining new approaches for extending operating system security.

File Systems: File system innovations began with BSD (the FFS introduced cylinder groups) and continue to this day. For example, SUJ adds journaled soft updates to UFS, and HAST provides synchronous, block-level replication over TCP/IP. Several FreeBSD developers are active in the OpenZFS community and this modern file system is built into FreeBSD.

Development Tools: The FreeBSD Project was an early adopter of Coverity Prevent and an early adopter of the CLANG/LLVM toolchain. FreeBSD provides built-in DTrace for system performance analysis.

In addition, the FreeBSD Project excels in several nontechnical areas:

License: The two-clause BSD license encourages the reuse of code. This allows the producers of commercial products to concentrate on their "secret sauce" and encourages the wide adoption of new standards.

Documentation: The Project provides many sources of documentation to help new users and new contributors to quickly get up to speed. These include the FreeBSD Handbook, FAQ, Developer's Handbook, Porter's Handbook, the Documentation Project Primer, and the built-in man pages.

Mentoring: The Project's processes and culture revolve around a mentoring environment. New code, documentation, and ports contributors are assigned several mentors who review and commit their patches as well as provide assistance for learning the Project's coding/documentation standards and best practices.

Community: As open-source projects go, the FreeBSD community is a friendly, encouraging, well-informed space. There is a wide range of expertise and experience levels and a culture of learning and professionalism.
Processes: As a mature (21 years) community, the FreeBSD Project is very organized and has well-documented processes. There are several organizational teams, including an elected core team to help set direction and resolve disputes, a security officer and security team to manage security advisories, a release engineer and release engineering team to manage software releases, a documentation engineering team to manage the documentation infrastructure, a port management team to manage the Ports Collection, and various infrastructure teams to manage the Project's websites, repositories, and other infrastructure.

How Do I Get Started?

Contributing code or documentation to FreeBSD can be as simple as finding something that interests/bugs you and submitting a patch that makes it better. This section elaborates on that statement by providing some helpful tips for interacting with the Project.

As an example, let's say you have found a typo in a man page or you have a patch that allows a broken port to successfully build. Go to bugs.freebsd.org, create a new account, and confirm the account-creation email. You can then use the quick links to create a bug report and attach your patch. More details on how to create a useful bug report can be found at freebsd.org/support/bugreports.html. In theory, someone will review your patch, possibly ask for more information, and commit the change for you. Once committed, your contribution is now "live" and able to benefit other FreeBSD users. Quick and painless!

What if you have grander ambitions than just a one-off patch that fixes some little thing you happened to stumble across? In this case, you want to start integrating yourself into the community. Typically this is done by subscribing to the mailing list(s) that match your interest(s). Depending upon the technical nature of the list and your comfort level, you might introduce yourself and your interests or just lurk for a while in order to get a good feel for the tone of the conversations and a better idea of who does what.

As you submit patches, don't get frustrated if they are not reviewed right away. There may be several reasons for this. If the patch is fairly long or complex, a developer needs to take the time to digest and understand the patch and how it fits into the larger scheme of things. If the patch is fairly simple, double-check that you have included all of the necessary information in your submission and that the utility of the patch is obvious and not too esoteric. If a week or so has gone by without any feedback, send a polite email to a related mailing list to inquire if anything else is needed to assist patch reviewers. Be sure to promptly respond with any information that is requested to help expedite the patch review process. Remember that your patches are being submitted to a community that consists of real people with busy lives and limited free time. They do share your passion for FreeBSD and want to work with others who respect their time and abilities.

Another contribution example is a company that would like to contribute back their patches. In this instance, the best case scenario is having at least one employee who either has a relationship with the FreeBSD community or who happens to be a FreeBSD developer. The worst case scenario is to dump a large chunk of code into a bug report—as any developer knows, it is very difficult and time-consuming to test and integrate large chunks of someone else's code into an existing codebase. If the organization does not have an existing relationship with the community and is interested in establishing one, the Foundation can assist in making introductions to the appropriate developers and possibly in arranging a visit with the organization's engineering and management teams to further discuss their collaboration goals.

As an individual, keep the following points in mind when interacting with the FreeBSD community:

Don't be shy. Your contributions matter. Find an area that interests you, submit good patches, interact with your reviewer, and learn from their suggestions.

Be aware that the community notices people who stick around, respect others, seem willing to learn, and continue to contribute. Don't be surprised if someone at some point asks if you would like to be mentored. This is a good thing.

Additional Resources

In summary, these resources cover a variety of types of contributions:

- Submit a code, port, or doc patch or report a bug: bugs.freebsd.org
- Submit event listings and news items: lists.freebsd.org/mailman/listinfo/freebsd-doc
- Assist new users: forums.freebsd.org
- Discuss the creation of a testimonial/case study or request company introductions or a Foundation visit by sending an email to board@freebsdfoundation.org
- Donate to the Foundation: freebsdfoundation.org/donate
- Include @freebsdblogs when Tweeting new FreeBSD-related blog posts
- Find additional documentation at freebsd.org/docs.html

Dru Lavigne has been using FreeBSD as her primary platform since 1997 and is the lead documentation writer for the FreeBSD-derived PC-BSD and FreeNAS projects. She is author of BSD Hacks, The Best of FreeBSD Basics, and The Definitive Guide to PC-BSD. She is founder and current Chair of the BSD Certification Group Inc., a nonprofit organization with a mission to create the standard for certifying BSD system administrators, and serves on the Board of the FreeBSD Foundation.

Testers, Systems Administrators, Authors, Advocates, *and of course* **Programmers** *to join any of our diverse teams.*

COME JOIN THE PROJECT THAT MAKES THE INTERNET GO!

★ DOWNLOAD OUR SOFTWARE ★

http://www.freebsd.org/where.html

★ JOIN OUR MAILING LISTS ★

http://www.freebsd.org/community/mailinglists.html?

★ ATTEND A CONFERENCE ★

• *http://www.bsdcan.org* • *http://www.asiabsdcon.org*

The FreeBSD Project

The FreeBSD Foundation

by Rick Miller and Julien Charbon

A Journey

From Adoption to Contribution

Selecting the right platform on which to deploy a product is fairly straightforward, in that it requires evaluating available options against a product's key metrics. And even deploying the platform is relatively trivial, barring the complexity of integrating it with systems like provisioning and installation frameworks. FreeBSD was able to meet our requirements, and thus was adopted and deployed as a diversity platform in the Domain Name System infrastructure operated by Verisign. But we didn't stop there.

To encourage further development of FreeBSD, we became—and remain—an active participant and collaborator within the community, which now includes one of our own engineers who has become a committer to the project.

The decision to adopt and deploy FreeBSD globally within Verisign's Domain Name System infrastructure was just the start of our involvement. We knew from the beginning that it was important—and necessary—to actively engage and collaborate with the FreeBSD Foundation, the project, and the support community. But before engaging with the community, it was important for us to understand the project and who operates it and to comprehend the ecosystem—how the components relate to one another and how they operate together.

The FreeBSD website (http://freebsd.org) contains valuable resources that describe the FreeBSD ecosystem in terms of philosophy, policy, processes, and practices. The functions of FreeBSD's core, release, and security teams are also well documented with detailed descriptions of the practices and processes for each. Additionally, the policies for communicating with, submitting patches to, and becoming a committer are found here. Understanding these functions and processes provides deeper insight into how the various components interact with one another.

Electronic Communication

Our communication with the FreeBSD project began via email on the various project mailing lists. The mailing list archives contain examples of collaborative efforts and include technical interactions like customizing and building release media and ports/packages, integration of FreeBSD into installation frameworks, driver functionality, network performance, and more.

An unexpected mode of communication arose early on while evaluating vendor hardware against FreeBSD. Although this mode was exercised only once, it provided results. A network card in the hardware failed to function as expected, and so we communicated the problem to the vendor, who subsequently put us in touch with an organization that was a heavy user of FreeBSD. This organization, which had committers on staff, assisted us in identifying an existing upstream patch, which mitigated the functional issue that was subsequently merged into internal development branches. While this proved helpful, we also realize it's unlikely to be exercised again due to the rarity of the scenario and also because we now have internal talent to address such issues on our own.

Technical Networking

Electronic communication is valuable, but in an increasingly well-connected world, technical networking still provides invaluable benefits. Networking occurs at specially targeted events such as local and/or regional user groups and conferences like the regional BSD user group, CapBUG, based in the Washington D.C./Baltimore region, and conferences like MeetBSD and BSDCan, based in the San Francisco metropolitan area and Ottawa, Canada, respectively. The benefits of participating in these events are numerous. They range from a casual conversation with a committer with whom one has been conversing via email or mailing list, to something as complex as sitting in a workgroup at a conference-hosted hacker lounge and collaboratively completing a project.

At MeetBSD 2012, our engineers experienced these benefits firsthand as they had the opportunity of meeting, in person, with various individuals they had become familiar with through mailing lists and email. The rapport between iXsystems and Verisign culminated in the establishment of Verisign's first-ever vBSDcon in 2013—a great way to actively give back to the community.

BSDCan is, undoubtedly, the "must-attend" BSD-related conference on the North American continent, and it was in 2013 that Verisign sent its first small delegation to attend the event. The experiences gained at MeetBSD were built upon at BSDCan, as it included invitations to attend the Vendor and Developer Summits, which precede some BSD-related conferences. Additionally, it put our engineers in direct contact with FreeBSD engineers who were leading projects in which Verisign had a specific interest—such as networking performance.

Verisign's Experience

From a development perspective, porting the Verisign core DNS infrastructure software stack onto FreeBSD was seamless. In large part, this was due to much of the prerequisite software being supported by default and its high level of POSIX compliance. The last step for FreeBSD validation as a diversity operating system was to run stability and performance tests. However, Verisign's workload is specific enough to drive unique issues within the operating system like kernel panics, driver crashes, scalability issues, and more.

A kernel panic within udp_input() was one of the first interesting issues encountered. Engineers designed and executed experiments implementing code to reliably reproduce the problem in order to devise a proper solution which was later submitted to the project through the bug tracking system [1]. It was imperative that we debug the problem and have confidence in its scope before opening the bug report. This expedited technical discussions with appropriate committers and engendered confidence in the abilities of Verisign engineers to diagnose and resolve the issues. Traits like thoroughness, patience, and perseverance in bug reports and mailing list communications attracted the attention of FreeBSD developers working in the same fields.

Continued development facilitated discovery of more interesting edge cases [2][3][4] and encouraged tighter collaboration with the FreeBSD project. The Developer's Summit prior to BSDCan 2013 was the ideal venue to describe and discuss our findings with those working on related technologies as well as with the broader community. The in-person collaboration at the Developer Summit and BSDCan enabled specifically targeted cooperation that resulted in an increased pace of the submission/review/acceptance life cycle for code and patches.

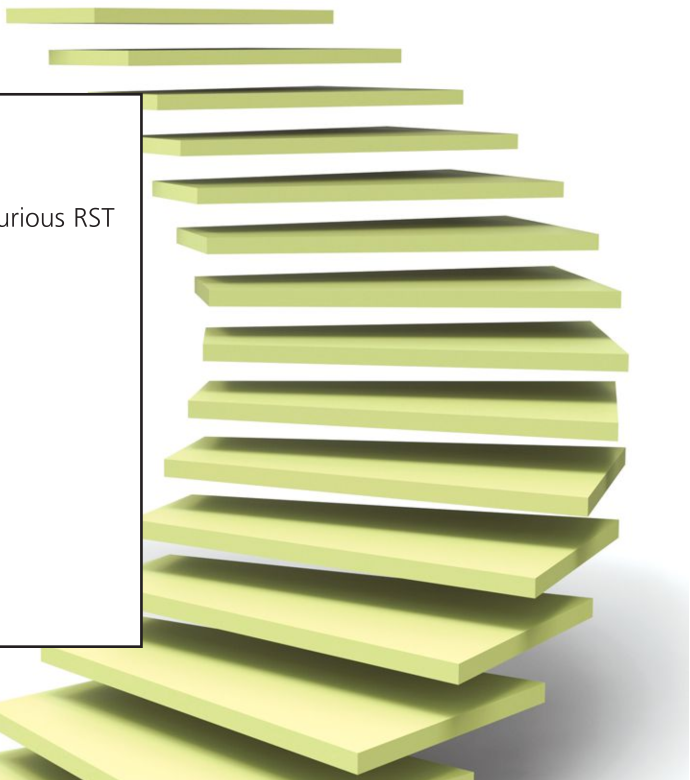
git, a well-designed tool that enables collaborative management of the full FreeBSD stack, was utilized to keep track of internal changes to FreeBSD sources using the official FreeBSD git mirror. This enables Verisign to build FreeBSD with patches for testing under real scenarios—prior to submitting them back to the project. Additionally, it helps maintain the relevancy of patches, particularly when review and patch improvement timelines don't line up. Furthermore, our relevant changes are tracked and published, making them available to the project for inclusion in their own development branches [5].

The associations continue to evolve, resulting in broader exposure of our work in the community through such avenues as the BSDnow podcast, a renewed invitation to the BSDCan 2014 DevSummit, and our article on a TCP network stack performance scalability issue and Verisign's proposed strategy to solve it in *FreeBSD Journal*'s May/June 2014 issue [6]. And, last but not least, one of Verisign's engineers [Julien Charbon] was welcomed into the community of FreeBSD committers.

From an email on freebsd-questions asking about customizing installation media to filing the first bug report to attending and organizing conferences to obtaining commit bit privileges, this journey illustrates the level of work and dedication that it takes to become involved in meaningful ways. It takes a great deal of time by dedicated engineers—from both organizations—sometimes performing work not directly related to their jobs. But it is done from a sense of contributing to a project that aims to provide the community a platform on which to achieve their own goals. We thank the FreeBSD project, its community, and the Foundation for their continued support, and we look forward to future collaborations, perhaps at the upcoming vBSDcon 2015 (to be formally announced at a later date). •

Julien Charbon is a software development engineer at Verisign Inc. Julien has worked on the company's high-scale network service ATLAS platform and related high-scale network services. Julien has worked with FreeBSD to perform tasks including porting software, developing kernel fixes and patches, and network stack performance improvements.

Rick Miller is a UNIX systems engineer at Verisign Inc., where he builds infrastructure systems supporting global DNS resolution platforms. Rick's focus over the previous five years has been integrating and deploying FreeBSD into these platforms. This includes building development/operational support systems and managing operating system source code and image builds.



References

[1] Kernel panic in udp_input()
https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=172963

[2] Concurrency in ixgbe driving out-of-order packet process and spurious RST
https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=176446

[3] TCP stack lock contention with short-lived connections
https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=183659

[4] Kernel panic: Sleeping thread owns a non-sleepable lock from netinet/in_multi.c
https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=185043

[5] Verisign's FreeBSD public repository
https://github.com/verisign/freebsd

[6] FreeBSD Journal May/June 2014 issue
https://mydigitalpublication.com/publication/?i=217642\

Help Grow FreeBSD

(Donate Today...)

Your donations over the past year have led to great progress for FreeBSD and the community.
Thank you!!

The Foundation is looking forward to continuing that progress in 2015. The areas we'd like to invest in include the following:

Funding Projects to Advance FreeBSD
Projects may include:

- Enhancements to the binary package build, distribution, and verification mechanism.
- Improving automated testing.
- Updated development and performance analysis tools.
- Support for the 64-bit ARM architecture.

Increasing Our FreeBSD Marketing Efforts

- Creating additional marketing collateral to recruit for FreeBSD
- Providing more marketing literature to educate on FreeBSD
- Helping grow our base of community investors
- Promoting the Project and The Foundation

Providing Conference Resources and Travel Grants

- Facilitating a greater number of FreeBSD representatives in attending conferences around the world to give presentations on FreeBSD.
- Providing FreeBSD materials in multiple languages

Support FreeBSD

Donate to the Foundation!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system. Our mission is to continue and increase our support and funding to keep FreeBSD at the forefront of operating system technology.

Thanks to people like you, the FreeBSD Foundation has been proudly supporting the FreeBSD Project and community for almost 15 years. We are incredibly grateful for all the support we receive from you and so many individuals and organizations that value FreeBSD.

Make a gift to support our work in 2015. Help us continue and increase our support of the FreeBSD Project and community worldwide!

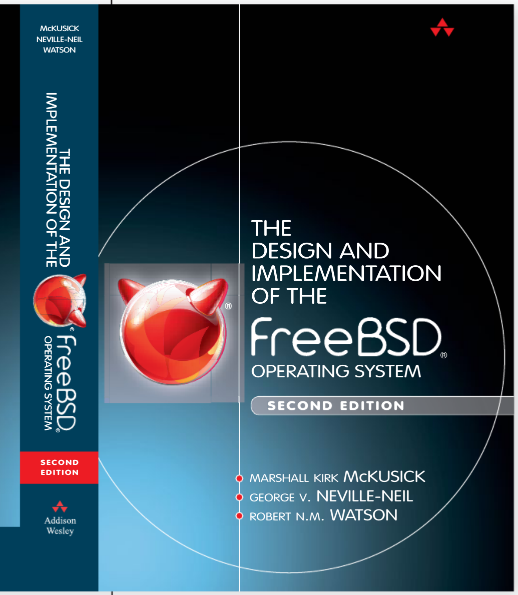
Making a donation is quick and easy. Go to:
www.freebsdfoundation.org/donate

The FreeBSD FOUNDATION

www.freebsdfoundation.org

BOOKreview

by Rik Farrow



The Design and Implementation of the FreeBSD Operating System, Second Edition

Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson

2015 Pearson Education, ISBN 978-0-321-96897-5 859 pages

This book comes out of a lineage of books about the BSD operating system, starting with the Design of 4.3BSD UNIX in 1989. While being focused on FreeBSD sets this book apart from other operating systems books where the focus is Linux, that's not the only thing that sets it apart.

Kirk McKusick has been involved in key design decisions that still have bearing on UNIX-related systems since he was a graduate student sharing an office with Bill Joy. And this book reflects not only McKusick's influence on the designs of file systems and virtual-memory systems, but also that of its two other authors.

Where a book like Robert Love's *Linux Kernel Development* dives into getting, building, and examining kernel code, *Design and Implementation* stays at a higher level. Algorithms and data structures are explained, but so are the design decisions behind why a particular algorithm or design was chosen.

Soft updates provide a particularly contentious example. Early Linux file systems could create and delete files much faster than the 4.3 BSD fast file system (FFS), because the authors of ext2 had decided to do away with ordered, synchronous writes of file system metadata. The FreeBSD developers' response, led by McKusick, was to create a process called soft updates, which allows metadata updates to occur asynchronously, but still in an ordered manner. Soft updates are explained in clear and concise text, both why they are considered necessary and how they need to work. Approaches that log metadata updates are considered in the following section (the approach used in Linux's ext3).

Like operating systems books in general, the book begins with a history of UNIX written by one of its participants, followed by an overview of the kernel. Process management follows, then a completely rewritten chapter on security. If you are seriously interested in operating system security features, this chapter provides an excellent overview of the many mechanisms that have appeared, and been implemented, over the past 25 years. While the Linux Security Module and the related SELinux and Type Enforcement get only brief mention, there are thorough discussions of access control lists, mandatory access control, the new NFSv4 ACLs, security event auditing, cryptographic services, random number generator, jails, and the Capsicum capabilities model, a recent addition to FreeBSD.

The next chapter, on memory management, is just as long as the security chapter, and just as detailed. The next part of the book covers the I/O system, starting with overview, then devices in general, moving to FFS, then a new chapter on the Zettabyte File System. Again, this chapter would be useful to anyone who wants a deep understanding of ZFS, whether you are using FreeBSD, or Solaris descendants like Illumos. The I/O section ends with a chapter on NFS, including NFSv4.

Part four covers Interprocess Communication, which begins with IPC and continues with chapters on network layer protocols, like IPv4 and IPv6, followed by transport layer protocols. The book concludes with a chapter on system startup and shutdown, and a glossary.

Each chapter ends with exercises and one or more pages of references. The exercises cover ideas from each chapter, as well, to help the dedicated reader think about potential solutions that go beyond what's covered in each chapter.

I did what I usually do with large technical books: I jumped around, after reading all of the introductory material, focusing on the parts I found most interesting. The writing makes this easy to do, in that I rarely found myself referred to another section in the book.

If you want or need to have a better understanding of how the FreeBSD operating system works, you should get this book. You can read all of it, or just the parts you need (once you've read

"If you want or need to have a better understanding of how the FreeBSD operating system works, you should get this book. You can read all of it, or just the parts you need."

the first section), and expect answers to your questions about why something is done in a particular way, such as the use of paging, IPv6 implementation, or virtualization with FreeBSD. And while each version of BSD is different, the versions do borrow a lot from each other, and roughly have the same design philosophy. What makes this book special is that there is a wealth of experience, a record of different approaches taken, written by three FreeBSD committers, all with stellar records. •

Rik Farrow is editor of *;login:*, the USENIX membership magazine. Rik began working with Unix in 1983 by writing systems documentation for microcomputer-based UNIX systems, then went on to write several books about Unix. His research into Unix security led him into teaching and consulting about UNIX and Internet security for many years. Along the way, he consulted as technical editor for *UnixWorld* magazine. Rik is also tutorial manager for USENIX.

FreeBSD

VPS HOSTING in the Cloud

- **Deploy in 30 Seconds**
- **24/7/365 Live Support**
- **No Contract, Cancel Anytime**
- **Free Setup, Free DNS**
- **100% Enterprise SSD**
- **1TB Outbound Transfer**
- **1 Gbps Port**

Plans start at only 99¢ per month

atlantic.net℠

GET STARTED TODAY!

conference REPORT

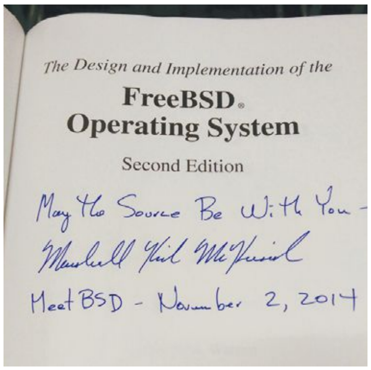
by Wallace Barrow

meetBSD

Going to any conference alone can be somewhat intimidating. Scheduling, traveling, and knowing no one can put you on edge. This was not the case for me going to meetBSD 2014. I could not wait to go to my first meetBSD conference, and I was greeted with open arms. Getting involved in a new community is always fun, as I discovered when I first started using FreeBSD three years ago.

meetBSD California 2014 was awesome! The night before the conference started I got to meet a handful of great people, and many more during the conference, including Dru Lavigne, Sean Bruno, Jim Maurer, Michael Dexter, and Glen Barber, just to name a few. Putting names to faces is a surreal experience.

I greatly enjoyed the breakout sessions and wished there was more time to attend them all. However, of all the discussions, I was most interested in bhyve. As a server administrator, I use Xen Server on a day-to-day basis, and learning about another hypervisor I could use in production is very intriguing. Talking to the developers directly and learning how this hypervisor works was the most beneficial thing a tech could ask for. You have direct access to the experts' knowledge, and that goes for any topic you are interested in at meetBSD. I hope to start some benchmarking of Bhyve in a lab I have set up very soon.



The Design and Implementation of the
FreeBSD® Operating System
Second Edition

May the Source Be With You –
[illegible]
MeetBSD – November 2, 2014

Presenters

We had a great two days of presenters. Rick Reed from WhatsApp showed us a great outline of a half a billion FreeBSD users. Currently WhatsApp holds over 600M monthly users, 140M concurrent connections, 440K connections per second, and 1.1M messages sent per second.

Jordan Hubbard gave us an overview of "FreeBSD: The Next 10 Years" in light of celebrating FreeBSD's 21st birthday on November 2, 2014. Jordan explained how we need to become more Lego-like in the architecture we create and choose hardware platforms that have mass appeal in the industry.

David Maxwell showed us a new Unix command called pipecut, with which users can create pipeline commands quickly and easily. We can create hotkeys stitching many Unix commands together to manipulate data and see each step of the process. I got to sit down with David at the social mixer and discuss how I will use this in my own environment, which was a very cool experience.

Kirk McKusick told us the story of IPWars. In summary: "If Kirk McKusick had a time machine, he'd put 48 bits on Ipv4" @Brendan Gregg on Twitter. I also got my new book, *The Design and Implementation of the FreeBSD Operating System,* Second Edition, signed by Kirk.

Brendan Gregg works for Netflix and showed us five major areas of performance analysis in FreeBSD: the tools, methodologies, benchmarkings, tracing, and counters. FreeBSD provides 34.2% of Internet traffic today because of Netflix.

If you have never been to a meetBSD conference or any BSD conference, start planning to do so and making arrangements now. From the people, the presenters, the breakout sessions, you get all you ever wanted from it. I am very excited to attend the next conference in Canada. See you all then! •

With a major in computer science and a minor in security, Wallace Barrow worked for the University of Wisconsin administrating a Windows Domain for an agriculture department. He began working with FreeBSD three years ago when he took a job with a small, local ISP in Madison, Wisconsin. Wallace loves his Packers, Badgers, and open source!

FreeBSD™ JOURNAL

LEAK THIS!

● BLOG IT ● TWEET IT ● READ IT

Available at amazon apps

Download on the App Store

ANDROID APP ON Google play

FreeBSD Journal is available as a mobile app for iPad, iPhone, Kindle, and all Android devices. Also available as a desktop Edition at www.freebsdfoundation.org. $19.99 per year (6 issues), or $6.99 per single copy.

FreeBSD™ Journal is published by The FreeBSD Foundation

this month In FreeBSD

BY DRU LAVIGNE

As a mature open-source project, the FreeBSD Project has a well-defined and well-documented process for managing releases, as well as a Release Engineering Team and Lead Release Engineer. This month, we had a chance to interview Glen Barber, who has been on the FreeBSD Release Engineering Team since January 2013 and lead Release Engineer since March 2014.

Q Tell us a bit about yourself. What is your technology background and how did you get started with FreeBSD?

A I'm 33 years old and currently live in Pennsylvania, USA. I've been using FreeBSD since around 2006, after reading about it in a Linux magazine. I remember sitting in the waiting room of a doctor's office and came across a small article about this thing called FreeBSD, which had just released 6.1-RC1. I folded over the corner of the page as a reminder to try FreeBSD when I got home from the appointment, mainly because I was looking for a different operating system to run on my Dell Inspiron b120 laptop.

At the time, I had just resumed going to college. I had been using Linux on my laptop for maybe one or two years, because it was easy to find different software and tools. Back then, I had no idea what I wanted to do for a career, but I knew I wanted to work with computers. This was when I first became aware of open-source software and started to discover all of the great projects and communities that existed.

So when I got home after the doctor's appointment, I downloaded the 6.1-RC1 installer, and installed my first FreeBSD system on a spare desktop-style computer. There were a number of things I liked about FreeBSD (and some that I did not like), but overall, something about it just felt right. In particular, I liked the separation between the base system and third-party software, which was one of the things I did not like about Linux—each distribution did their "base install" differently, and, of course, each with a different utility. The ability to run a few commands and end up with just the base installation was one of the things I found most attractive about FreeBSD.

Once I was comfortable enough with FreeBSD, I replaced the Linux installation on my laptop with 6.1-RELEASE, and since then, every computer I've owned (laptops, desktops, and servers) has all been FreeBSD systems.

In hindsight, I'm often amused at the almost accidental chain of events that led to my involvement in FreeBSD, especially considering my roles within the Project.

Q Most readers will already have some idea of how the release engineering process works. What is the typical workday like for a Release Engineer? How does that work day evolve as a release date draws near?

A The FreeBSD release cycle actually starts several months before any publicly-visible changes take place, such as the code freeze or release candidate builds.

Lately, we've tried to have the schedule for the next release written and agreed upon within the Release Engineering Team three months prior to the start of the cycle, after which we'll send several reminder emails to the FreeBSD developers.

While this may seem trivial, it's quite important to communicate the target dates with FreeBSD developers very early because we typically get a number of replies informing of work that is in progress that is targeted to be included in the release, such as new features or non-trivial bug fixes. In some cases, we will grant blanket approvals to developers, so pre-commit approval is not required for each change when the code freeze is in effect. Granting blanket approval depends on several factors, such as the scale of the targeted changes and how self-contained the changes are expected to be. For example, something like a new network device driver would likely be a candidate for granting a blanket approval, while a major overhaul of the virtual memory subsystem would not.

Writing the schedule for an

operating system release is a bit more complex than one might imagine, because there are so many different moving pieces that need to all be coordinated and timed perfectly for everything to fit together. For example, we need to make sure we have binary packages built for the release, which cannot be done until we get close to the end of the release cycle. The FreeBSD Documentation Project sources need to be tagged for the release, so the documentation packages are available for the release, and this needs to happen before the package builds. Since the FreeBSD Ports Collection started branching the tree quarterly, it also makes sense to align the package builds with the quarterly branch, which is also the basis of the release tag.

Then, each phase of the release cycle, such as BETA and RC builds, needs to be timed properly. For 10.1-RELEASE, each BETA and RC build took around nine hours, from start to finish for all of the supported architectures. If anything goes wrong during the builds (either from human error or a bug in the build process), the builds need to be restarted from the beginning. Then it takes roughly another eight hours for the images to propagate to the FTP mirrors. Then there needs to be enough time allocated for testing, both by the Release Engineers and FreeBSD consumers, and a sufficient amount of time for bug fixes before the next builds begin. This all needs to be considered months in advance when writing the schedule, and as you can see, makes it very easy to fall behind schedule. It's surprisingly complex. It is really like putting together a puzzle, in a way.

Once the code freeze starts, things begin to pick up quickly. Each change during the code freeze requires explicit review and approval from a member of the Release Engineering Team, unless a blanket approval is granted. There is a lot of email during the release cycle, but luckily there are a number of highly active FreeBSD developers on the Release Engineering Team, so the turnaround time for the review/approval process is quite low. Depending on the change in question, we'll occasionally defer to someone within Release Engineering who has expertise in the part of the system for the change review, or request review from someone outside of Release Engineering, which may delay a response to the approval request, but it is not usually too much of a delay.

About a day before the start of a BETA or RC build, I will send an email to the Release Engineering Team to inform that the builds will begin as planned, and any incoming commit approval requests should not be answered until after the builds are completed and have been tested. I'll also send a similar email to the FreeBSD developers to inform them that any approval requests will be answered after the builds have finished, so developers do not think their request was not received.

Lately, I've been trying to start the builds just after 00:00 UTC, but this isn't a defined part of our work flow or policies. The main reason for this is the length of time the builds take, and the timing based on the order in which the builds complete. Since FreeBSD 9.2-RELEASE, release builds for non-x86 architectures have been cross-built on amd64 hardware, which makes the order of completion predictable. Once the builds are started around 00:00 UTC (about 20:00 local time), I can be fairly certain that the FreeBSD/amd64 and FreeBSD/i386 builds will finish around midnight, after which I copy the installation medium to my local test machines. This usually takes a few hours, so I'll get a few hours sleep while the remaining builds finish running in serial, and the x86 builds finish copying locally.

Of course, this is assuming there aren't any problems. When things do go wrong, they need to be addressed. The problems can vary and do not necessarily indicate a problem with the FreeBSD source tree. There are human factors as well. On more than one occasion (more times than I'd like to admit), I have had to restart BETA, RC, and, yes, even final RELEASE builds because I've made a mistake. These mistakes can range from setting the incorrect Subversion revision for the tree in the release.conf configuration file for the build, to a subtle typographical error in a configuration file. Mistakes happen, and when they do, they guarantee a sleepless night.

> “The evolution of any given FreeBSD release is interesting because there are so many different factors involved.”

Once the builds finish, I copy the installation images to a public web server, and send a PGP-signed email to the FreeBSD Release Engineering Team and FreeBSD Security Team, providing the URL for the images where others on the Release Engineering Team can download and help with testing. The Security Team then starts building the various sets of patches for freebsd-update(8) consumers.

Once the images are tested, they are uploaded to the primary FTP mirror, pending propagation to the various worldwide mirrors. I'll then send an email to the Release Engineers and Security Team to give an approximate time frame of when the announcement email will be sent to the mailing lists. After the announcement email is sent, we start the process over again by replying to the commit approval requests that have been pending since the builds started, and keeping a close eye on the mailing lists and problem report system for issues that may arise.

The evolution of any given FreeBSD release is interesting because there are so many different factors involved. The influx of commit approval requests at the start of the release cycle is fairly steady, but as we progress through each of the various stages of the release cycle, we need to be increasingly more cautious about what changes we will allow. Conversely, as the cycle progresses and BETA and RC builds are released, there seems to be a slow, but steady increase in adoption of what will be the next release.

I should note that we don't have any hard evidence to support this other than my own observations relating to the stage of the release cycle and problem reports. As we get closer to the end of the release cycle, we need to be extremely cautious as to what change requests are approved. This becomes particularly challenging at the end of the release cycle, because we do want to fix outstanding issues with the release; however, there is always the potential of introducing a regression, which could make the last few weeks of the release cycle particularly stressful.

“Being the Release Engineer for an operating system is tough, and probably not comparable to any other software project.”

As an example, imagine there is a problem report about a disk controller driver that scribbles garbage to the drive every 300th write. It's a serious problem, and certainly behavior that is far from desirable. As FreeBSD Release Engineers, the solution is much less simple than "just fix it," especially when we have already announced what was, according to the schedule, the final release candidate. At that point of the cycle, every solution is non-ideal.

A driver update will make it necessary to add another release candidate to the schedule. But what if it isn't such an easy fix as a driver update? What if the fix requires an update to the cam(4) subsystem? Will the fix adversely affect other drivers using cam(4)? Are we even sure the problem is not due to malfunctioning hardware and it isn't a bug at all? These are all questions that need to be answered, and their answers each play a significant role in what, ultimately, will be a difficult decision to the question "What do we do about this?"

Q What is the checklist of items that needs to happen before a release can be released?

A In addition to what was mentioned earlier regarding the order in which events need to take place when preparing the release schedule, the primary thing that needs to be done is testing. While this may seem like an obvious answer, there are a few things that cannot be tested until a prerequisite stage has happened, so each stage of the release cycle may differ slightly.

For example, when BETA1 builds are complete, I tend to focus my attention on the installer and other "setup" class software in the base system. A problem in bsdinstall(8) can quickly change my focus for BETA2 if a serious problem is found. My primary testing environment consists of a relatively modern, powerful machine with a significant amount of disk space (4.2 TB free space right now) running FreeBSD-CURRENT. I use VirtualBox to test the installation images and installed system because creating machines with varying configurations can be scripted.

I start with three provisioned virtual machines, one for FreeBSD/amd64, one for FreeBSD/i386, and recently added to the setup, one for FreeBSD/amd64 with UEFI. I take these provisioned machines, each already configured with virtual hard drives, allocated RAM, and virtual CPUs, and clone them for various different installation paths. I'll create a "FreeBSD amd64 bootonly" virtual machine to boot-test the bootonly.iso image, but since the distribution sets do not yet exist on the FTP mirrors, I cannot do much more with those images once they are verified to boot. We do not publish the images on the FTP mirrors until they are tested, in case we find a critical problem that makes it necessary to redo the build, in which case it would be more difficult to pull the problematic builds from the mirror sites.

For the disc1.iso installer image, I create four independent virtual machines. One for testing the installation, using the default for all options; one for testing the installation using a root-on-ZFS setup; one for testing root-on-ZFS with GELI encrypted disks; the last one is a clone of the first. When the installation is complete, I install a GNOME desktop on the first virtual machine and a KDE on the second, and do basic testing. Since the finalized package sets are far off in the future, it does not make sense to spend too much time doing in-depth testing of the desktop machines just yet, given how quickly the ports tree changes. What does not work today may work tomorrow, and vice-versa.

Then I create two virtual machines for the dvd.iso installer, one for GNOME and one for KDE. Since I have already tested the installation paths using the disc1.iso, I don't need to spend too much time on the installer at this point, and can focus on testing package installation from the packages included on the DVD itself.

When the BETA2 builds are ready in the next phase of the release cycle, I use the change list from the subversion logs as the focus of the testing. Assuming there were not any changes to the installer since BETA1, I can focus less on the details of the various installation paths, and spend more time testing (as much as possible, given hardware limitations) what has changed since BETA1.

When the first release candidate is ready, the FreeBSD Ports Management Team can do the package builds for the release. The caveat here is if there is a change that affects either the ABI (Application Binary Interface) or API (Application Programming Interface) in FreeBSD after the package builds have finished, they must be rebuilt. Unfortunately, this does happen.

We generally have the finalized package set available for RC2, and at that point I start to focus the testing more on using the system, such as testing various parts of GNOME and KDE. I'll also install various server software, such as Apache, Nginx, and PostgreSQL, to make sure there are no glaringly-obvious problems.

The benefit, I think, of doing testing this way is that I can worry less about installation issues near the end of the release cycle and focus attention on what we expect to be the released operating system.

Q The tasks performed by a Release Engineer are just one part of a very large project. Who else does the Release Engineering Team work with?

A We work very closely with the FreeBSD developers, mainly through the review/approval process, and the Ports Management and Documentation Engineering Teams to coordinate tagging the various source trees and package builds.

However, I would have to say that we work most closely with the FreeBSD Security Team, and for a number of reasons. The FreeBSD Security Officer and Deputy Security Officer are the two people that have access to the freebsd-update(8) build machines, so during the BETA and RC phases we need to keep our communication channels open. The Release Engineering Team and Security Team need to be in sync in various other parts of the process, such as determining the end-of-life date for the release, and following up on any outstanding errata items after the release.

Q You have been the Lead Release Engineer for almost a year now. What have you learned? Any surprises along the way?

A It has been a learning experience, that's for sure. I certainly learned how difficult the job of being the Release Engineer is. I truly cannot express my respect and gratitude to all of the FreeBSD Release Engineers enough.

Being the Release Engineer for an operating system is tough, and probably not comparable to any other software project. And, to be clear, I don't at all mean that other software projects are not as important as an operating system. But, when releasing an operating system, there really is great potential for disaster.

It is, without a doubt, the most difficult job I've ever had. It is also the most rewarding job I've ever had, and an experience unlike any other. It's one of the reasons I'm truly grateful for the FreeBSD developers and FreeBSD community. We have amazing people in the FreeBSD Project.

You can learn more about the FreeBSD Release Engineering Team and its release engineering process at https://www.freebsd.org/releng/.

Dru Lavigne is a Director of the FreeBSD Foundation and Chair of the BSD Certification Group.

svnUPDATE

by Glen Barber

IT'S THAT TIME AGAIN—a new year and a great time to take a look at some of the new features and enhancements that FreeBSD developers have been working on. Sit back, relax, and enjoy the highlights.

Whether you are a FreeBSD systems administrator, application developer, or hobbyist, changes to the system, in particular utilities and configurations, are some of the most important things to keep an eye on. Here are some changes that can be expected in upcoming FreeBSD releases.

The mailwrapper(8) Utility

http://svnweb.freebsd.org/changeset/base/270675

The mailwrapper(8) utility is used to invoke the appropriate MTA (Mail Transfer Agent) such as Sendmail, Postfix, or qmail based on what is specified by the configuration file mailer.conf(5). The mailer.conf(5) file is used to map certain commands used by the mail subsystem to the absolute path of the program that should be used for the command.

For example, mailer.conf(5) may contain commands such as 'sendmail', 'mailq', 'newaliases', and 'hoststat'. On FreeBSD, these commands map to the /usr/libexec/sendmail/sendmail program, which is based on the command name that is used to invoke it, and will behave differently to modify parts of the email system. newaliases(8) will rebuild the aliases(5) database file aliases.db; hoststat(8) will print the host status database used during SMTP transactions, etc.

Historically, mailer.conf(5) would exist in the base system configuration directory, /etc/mail, but as of revision 276917 it is now possible to put the mailwrapper(8) configuration file elsewhere on the system, avoiding conflicts and/or inadvertent reversal of local changes. The mailwrapper(8) utility now respects the LOCALBASE environment variable, and will search there first for its configuration file, which overrides the FreeBSD base system default mailer.conf(5) if it exists. LOCALBASE, in FreeBSD nomenclature, is the root of the non-base installation path, /usr/local.

The rc(8)Subsystem

http://svnweb.freebsd.org/changeset/base/276918

The rc(8) subsystem is one of the most essential subsystems in FreeBSD used to determine whether or not a service is started at boot, and also for command-line arguments that should be used in conjunction when the service is started.

Although the rc(8) subsystem is arguably one of the most complex pieces of FreeBSD, its configuration has remained remarkably simple for administrators, and its simplicity and extensibility has had another update. The rc(8) subsystem now has support for allowing service startup configuration files in both /etc/rc.conf.d and LOCALBASE/etc/rc.conf.d (which expands to /usr/local/etc/rc.conf.d by default), allowing administrators to create smaller configuration files for system configuration.

Each file within the rc.conf.d directory could contain, for example, the type of service which then would contain the configuration parameters for the service. One popular example of this is creating a /etc/rc.conf.d/jail file to contain all jail(8) startup parameters, and a /usr/local/etc/rc.conf.d/postfix file, which contains startup parameters for the Postfix mail server.

The linux(4) ABI Layer

http://svnweb.freebsd.org/changeset/base/271982

After much work, much testing, and even more necessity, the linux(4) ABI (Application Binary Interface) layer has been updated from Fedora Core 10 support to CentOS 6 support, allowing a number of recent, Linux-only applications to run on FreeBSD. To check the Linux ABI compatibility version on a system, examine the output of the compat.linux.osrelease sysctl(8):

- 2.6.16: Fedora Core 10
- 2.6.18: CentOS 6

If it is necessary to continue using the Fedora Core 10 ABI layer, the default can be rolled back by adding 'compat.linux.osrelease=2.6.16' to sysctl.conf(5).

The bsdinstall(8) Installer

http://svnweb.freebsd.org/changeset/base/272274

The bsdinstall(8) installer is the default installation utility since the FreeBSD 9.0-RELEASE. Since the 9.0-RELEASE, bsdinstall(8) has received various updates and enhancements to supplant its successor, sysinstall(8).

The change in revision 272274 is important for both systems administrators and developers alike, especially when using root-on-ZFS installations—the default /var dataset now has the 'canmount=off' ZFS property set, so although /var is created by default, it is not automatically mounted at boot. This allows the system to have multiple /var datasets, which, when used with multiboot environments such as the sysutils/beadm port, prevent the /var directory from numerous individual environments from conflicting with each other.

There are a large number of very important files within /var. Everything from the pkg(8) database, to backup files generated from periodic(8) runs, to crontab(5) files, to system crash dumps exist there. This makes it very important to ensure that numerous independent environments that share nothing more than the physical hardware on which they run (and the underlying utility to control which environment boots by default) do not share this data, so one environment does not collide with the other, such as installed packages.

The crypto(4) Driver

http://svnweb.freebsd.org/changeset/base/275732

The crypto(4) driver, part of the OpenCrypto framework, supports two new cryptographic modes—AES-ICM and AES-GCM. The crypto(4) driver provides userspace access to hardware-accelerated cryptography devices and is used in conjunction with various other cryptography devices, such as aesni(4), ipsec(4), padlock(4), and random(4).

The OpenCrypto framework updates were sponsored by The FreeBSD Foundation.

The vxlan(4) Device

http://svnweb.freebsd.org/changeset/base/273331

The vxlan(4) device was recently added to FreeBSD which, analogous to vlan(4), creates a virtual tunnel endpoint. The vxlan(4) segment is a virtual Layer 2 network overlaid in a Layer 3 endpoint, which, compared to vlan(4), is better suited for multi-tenant data center environments.

The gre(4) Driver

http://svnweb.freebsd.org/changeset/base/274246

The gre(4) driver, which provides a generic routing encapsulation interface, has been split into two separate modules—gre(4) and me(4). The gre(4) module itself has been overhauled significantly, with its counterpart, me(4), providing an interface for more minimal encapsulation within Layer 3.

THANK YOU, READERS, for supporting the FreeBSD community, *FreeBSD Journal*, and, of course, The FreeBSD Foundation.

Don't forget, development ISO and preinstalled virtual machine images (in VHD, VMDK, QCOW2, and RAW formats) of the FreeBSD-CURRENT and FreeBSD-STABLE branches can be found on the FTP mirrors, and are built weekly: ftp://ftp.freebsd.org/pub/FreeBSD/snapshots/.

As always, development snapshots are not intended for production use; however, we do encourage regular testing so we can make the next FreeBSD releases as great as you expect them to be.

As a hobbyist, Glen Barber became heavily involved with the FreeBSD project around 2007. Since then, he has been involved with various functions, and his latest roles have allowed him to focus on systems administration and release engineering in the Project. Glen lives in Pennsylvania, USA.

PORTSreport

by Frederic Culot

In November and December the activity on the ports tree was not high, mainly due to end-of-year celebrations. Still, the figure is impressive with 3,778 commits on the ports tree! On the bug front, however, more problem reports were closed than during last period, with 1,157 problems fixed. Thanks to all of you who contributed feedback and took the time to make FreeBSD ports better!

IMPORTANT PORTS UPDATES

Several exp-runs were performed (22 actually) to check whether major ports updates are safe or not. Among those important updates, we mention the following highlights:

- llvm and clang updated to 3.5.0
- xorg-server updated to 1.14
- gnome updated to 3.14.2
- enlightenment updated to 0.19.2
- pkg updated to 1.4
- default Perl version set to 5.18
- default PostgreSQL version set to 9.3

As is usual, if manual steps are needed to update specific ports, then those steps are clearly mentioned in the /usr/ports/UPDATING file. It is highly advised to check this file before performing any update on the ports tree!

Also, please note that default versions of ports having multiple versions available are set in the /usr/ports/Mk/bsd.default-versions.mk file. If for some reason the default version of a port does not suit your needs, then you can simply override it by adding the DEFAULT_VERSIONS variable in make.conf as follows:

- DEFAULT_VERSIONS= perl5=5.16 ruby=1.9

NEW PORTS COMMITTERS AND SAFEKEEPING

During the last two months, several developers who already owned an src commit bit decided to join our community and were granted a ports bit, too. Those are: jmg@, jmmv@, and truckman@. Also, Muhammad Rahman was granted a ports commit bit and will be mentored by marino@ and bapt@.

Only one commit bit was taken in for safekeeping during the last two months (motoyuki@), but we also received sad news from miwi@, who decided to step down from duties at FreeBSD to focus more on his family and professional life. Unless you have been hiding under a rock for the past 10 years, you have likely heard about Martin (miwi@) Wilke. Simply put, Martin is the developer who has contributed the largest number of commits to the ports tree, with more than 20,000 commits since 2006!

YEARLY figures

2014 was the year that saw the highest number of commits in all of our ports tree's history! Whereas we never reached more than 30,000 commits before, 2014 saw almost 37,500 commits applied to our ports [https://people.freebsd.org/~eadler/datum/ports/commits_by_year.png]. So we take this opportunity to thank all our developers and contributors, and we hope to see such dedication in 2015 as well!

NEW portmgr LURKERS

Every four months the port management team welcomes a new pair of lurkers, i.e., two ports committers who are given an opportunity to contribute at a higher level, learn the inner workings of portmgr@, and to share in the workload. Those two lurkers were added to the portmgr@ mailing list, and will have access to confidential correspondence. They are encouraged to participate in all discussions and add their voice to the outcome of decisions taken by portmgr@.

A new term began in November, and our two lurkers are now ak@ and sunpoet@. As a tradition, they were asked to answer a questionnaire to get to know them a bit better. Here are links to the questionnaires answered by Alex [http://blogs.freebsdish.org/portmgr/2014/11/04/getting-to-know-your-portmgr-lurker-ak/] and Po-Chuan [http://blogs.freebsdish.org/portmgr/2014/12/03/getting-to-know-your-portmgr-lurker-sunpoet/]

Frederic Culot has worked in the IT industry for the past 10 years. During his spare time he studies business and management and just completed an MBA. Frederic joined FreeBSD in 2010 as a ports committer, and since then has made around 2,000 commits, mentored six new committers, and now assumes the responsibilities of portmgr-secretary.

THE TECHNICAL BSD CONFERENCE.
The BSD Conference held in Ottawa, Canada, has quickly established itself as the technical conference for people working on and with 4.4BSD based operating systems and related projects. The organizers have found a fantastic formula that appeals to a wide range of people from extreme novices to advanced developers.

BSDCAN 2015

COME JOIN US AT THE 12TH ANNUAL BSDCAN!

WHERE ....Ottawa, Canada
University of Ottawa

WHEN ....Thurs. & Fri. June 10-11 (tutorials)
Fri. & Sat. June 12-14 (conference)

BSD Certification • Courseware DVD • Register for an Exam
Join the growing ranks of people taking the BSDA exam.
If you missed it in the past, now is your chance to catch up.

GO ONLINE FOR MORE DETAILS
http://www.bsdcan.org

WELCOME

to AsiaBSDCon 2015!

DATE
March 12-15, 2015

LOCATION
Tokyo University of Science,
Tokyo, Japan
www.http://asiabsdcon.org

CONTACT secretary@asiabsdcon.org

THROUGH JUNE 2015

BY DRU LAVIGNE

Events Calendar

The following BSD-related conferences are scheduled during the first half of 2015. More information about these events, as well as local user group meetings, can be found at **www.bsdevents.org.**

SCALE • Feb. 19–22 • Los Angeles, CA

http://www.socallinuxexpo.org/scale13x/ • The 13th annual Southern California Linux Expo will once again provide several FreeBSD-related presentations, a FreeBSD booth in the expo area, and an opportunity to take the BSDA certification exam. This event requires registration at a nominal fee.

AsiaBSDCon • March 12–15 • Tokyo, Japan

http://2015.asiabsdcon.org/ • This is the annual BSD technical conference for users and developers on BSD-based systems. It provides several days of workshops, presentations, a Developer Summit, and an opportunity to take the BSDA certification exam in either English or Japanese. Registration is required for this event.

LinuxFest Northwest • April 25 & 26 • Bellingham, WA

http://linuxfestnorthwest.org/2015 • This is the 16th year for this annual, community-based conference. This event is free to attend and always has at least one BSD booth in the expo area and at least one BSD-related presentation. It also provides an opportunity to take the BSDA certification exam.

BSDCan • June 10–13 • Ottawa, Canada

http://www.bsdcan.org/2015/ • The 12th annual BSDCan will take place in Ottawa, Canada. This popular conference appeals to a wide range of people from extreme novices to advanced developers of BSD operating systems. The conference includes a Developer Summit, Vendor Summit, Doc Sprints, tutorials, and presentations. The BSDA certification exam and the beta exam for the BSDP will also be available during this event.

Are you aware of a conference, event, or happening that might be of interest to *FreeBSD Journal* readers?

Submit calendar entries to editor@freebsdjournal.com.

THE INTERNET NEEDS YOU

GET CERTIFIED AND GET IN THERE!

Go to the next level with BSD CERTIFICATION

Getting the most out of BSD operating systems requires a serious level of knowledge and expertise

NEED AN EDGE?

BSD Certification can make all the difference.

Today's Internet is complex. Companies need individuals with proven skills to work on some of the most advanced systems on the Net. With BSD Certification

YOU'LL HAVE WHAT IT TAKES!

SHOW YOUR STUFF!

Your commitment and dedication to achieving the **BSD ASSOCIATE CERTIFICATION** can bring you to the attention of companies that need your skills.

BSDCERTIFICATION.ORG

Providing psychometrically valid, globally affordable exams in BSD Systems Administration

FreeBSD™ Journal is published by The FreeBSD Foundation